1985-12

The implementation of a network CODASYL-DML interface for the multi-lingual database system.

Emdi, Bulent

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE IMPLEMENTATION OF A NETWORK CODASYL-DML INTERFACE FOR THE MULTI-LINGUAL DATABASE SYSTEM

by

Bulent EMDI

December 1985

Thesis Advisor: David K. Hsiao

ITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

PORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
CURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
ECLASSIFICATION / DOWNGRADING SCHEDULE	
RFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
AME OF PERFORMING ORGANIZATION al Postgraduate School	6b. OFFICE SYMBOL (If applicable) 52
7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
DDRESS (City, State, and ZIP Code) terey, California 94943-5100	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5100
AME OF FUNDING / SPONSORING RGANIZATION	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
DDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

TLE (Include Security Classification)
E IMPLEMENTATION OF A NETWORK CODASYL-DML INTERFACE FOR THE MULTI-LINGUAL DATABASE SYSTEM (UNCLASSIFIED)

RSONAL AUTHOR(S)
IDI, Emdi

YPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
ter's Thesis	FROM _____ TO _____	1985 December 19	127

PPLEMENTARY NOTATION

COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
ELD	GROUP	SUB-GROUP	Multi-lingual Database System (MLDS), Multi-backend Database System (MBDS), Network Data Model, Data Language CODASYL-DML, Attribute-based Data (Cont)

STRACT (Continue on reverse if necessary and identify by block number)

aditionally, the design and implementation of a conventional database stem begins with the selection of a data model, followed by the specification of a model-based data language. An alternative to this traditional proach to database system development is the multi-lingual database system (MLDS). This alternative approach affords the user the ability to access and manage a large collection of databases via several data models and eir corresponding data languages.

this thesis we present the specification and implementation of an interface which translates CODASYL-DML data language calls into attribute-based data language (ABDL) requests. We describe the software engineering pects of our implementation and an overview of the four modules which mprise our CODASYL-DML language interface.

STRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
AME OF RESPONSIBLE INDIVIDUAL id K. Hsiao	22b. TELEPHONE (Include Area Code) 408 646-2253
22c. OFFICE SYMBOL 52Hq

ORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Block # 18 (Continued)

Language (ABDL), Language Interface

Approved for Public Release, Distribution Unlimited.

The Implementation of a Network CODASYL-DML Interface for the Multi-Lingual Database System

by

Bulent Emdi
Ltjg, Turkish Navy
B.S., Turkish Naval Academy, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1985

ABSTRACT

Traditionally, the design and implementation of a conventional database system begins with the selection of a data model, followed by the specification of a model-based data language. An alternative to this traditional approach to database system development is the multi-lingual database system (MLDS). This alternative approach affords the user the ability to access and manage a large collection of databases via several data models and their corresponding data languages.

In this thesis we present the specification and implementation of a network CODASYL-DML language interface for the MLDS. Specifically, we present the specification and implementation of an interface which translates CODASYL-DML data language calls into attribute-based data language (ABDL) requests. We describe the software engineering aspects of our implementation and an overview of the four modules which comprise our CODASYL-DML language interface.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. MOTIVATION

During the past twenty years database systems have been designed and implemented using what we refer to as the traditional approach. The first step in the traditional approach involves choosing a data model. Candidate data models include the hierarchical data model, the relational data model, the network data model, the entity-relationship data model, and the attribute-based data model to name a few. The second step specifies a model-based data language, e.g., DL/I for the hierarchical data model, or Daplex for the entity-relationship data model.

A number of database systems have been developed using this methodology. For example, IBM introduced the Information Management System (IMS) in the sixties, which supports the hierarchical data model and the hierarchical-model-based data language, Data Language I (DL/I). Sperry Univac introduced the DMS-1100 in the early seventies, which supports the network data model and the network-model-based data language, CODASYL Data Manipulation Language (CODASYL-DML). And more recently, there has been IBM's introduction of the SQL/Data System which supports the relational model and the relational-model-based data language, Structured English Query Language (SQL). This traditional approach to database system development has resulted in homogeneous database systems that restrict the user to a single data model and a specific model-based data language.

An unconventional approach to database system development, referred to as the *Multi-lingual Database System* (MLDS) [Ref. 1], alleviates the aforementioned restriction. This new system affords the user the ability to access and manage a large collection of databases via several data models and their corresponding data languages. The design goals of MLDS involve developing a system that is accessible via a hierarchical/DL/I interface, a relational/SQL interface, a network/CODASYL/DML interface, and an entity-relationship/Daplex interface.

There are a number of advantages in developing such a system. Perhaps the most practical of these involves the reusability of database transactions developed on an existing database system. In MLDS, there is no need for the user to convert a transaction from one data language to another. MLDS also permits the running of database transactions written in different data languages. Therefore, the user does not have to perform either manual or automated translation of existing transactions in order to execute a transaction in the MLDS. MLDS provides the same results even if the data language of the transaction originates at a different database system.

A second advantage deals with the economy and effectiveness of hardware upgrade. Frequently, the hardware supporting the database system is upgraded because of technological advancements or system demand. With the traditional approach, this type of hardware upgrade has to be provided for all of the different database systems in use, so that all of the users may experience system performance improvements. This is not the case in MLDS, where only the upgrade of a single system is necessary. In MLDS, the benefits of a hardware upgrade are uniformly distributed across all users, despite their use of different models and data languages.

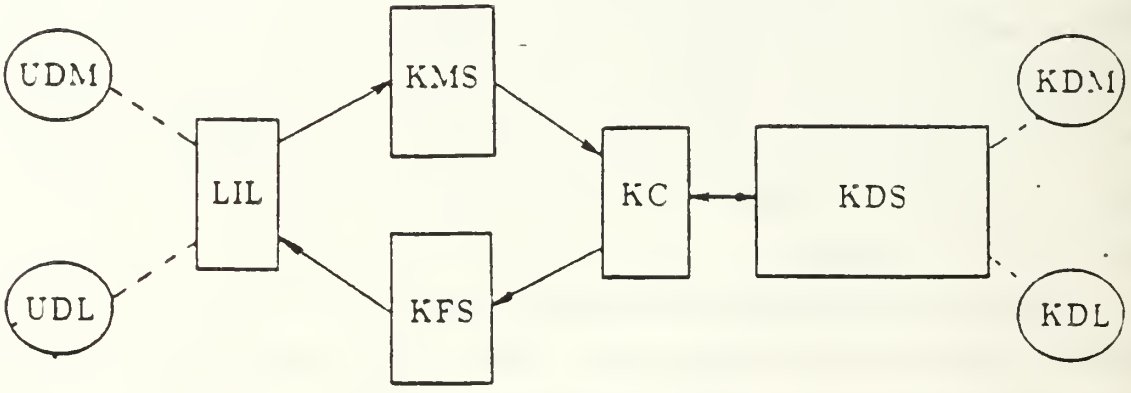
Thirdly, a multi-lingual database system allows users to explore the desirable features of the different data models and then use these to better support their applications. This is possible because MLDS supports a variety of databases structured in any of the well-known data models.

It is apparent that there exists ample motivation to develop a multi-lingual database system with many data model/data language interfaces. In this thesis, we are developing a network(CODASYL) language interface for the MLDS. We are extending the work of Banerjee [Ref. 2] and Wortherly [Ref. 3], who have shown the feasibility of this particular interface in a MLDS.

B. THE MULTI-LINGUAL DATABASE SYSTEM

A detailed discussion of each of the components of the MLDS is provided in subsequent chapters. In this section we provide an overview of the organization of MLDS. This assists the reader in understanding how the different components of MLDS are related.

Figure 1 shows the system structure of a multi-lingual database system. The user interacts with the system through the *language interface layer (LIL)*, using a chosen *user data model (UDM)* to issue transactions written in a corresponding model-based *user data language (UDL)*. The LIL routes the user transactions to the *kernel mapping system (KMS)*. The KMS performs one of two possible tasks. First, the KMS transforms a UDM-based database definition to a database definition of the *kernel data model (KDM)*, when the user specifies that a new database is to be created. When the user specifies that a UDL transaction is to be executed, the KMS translates the UDL transaction to a transaction in the *kernel data language (KDL)*. In the first task, KMS forwards the KDM data definition to the *kernel controller (KC)*. KC, in turn, sends the



UDM : User Data Model
UDL : User Data Language
LIL : Language Interface Layer
KMS : Kernel Mapping System
KC : Kernel Controller
KFS : Kernel Formatting System
KDM : Kernel Data Model
KDL : Kernel Data Language
KDS : Kernel Database System

Figure 1. The Multi-Lingual Database System.

KDM database definition to the *kernel database system (KDS).* When KDS is finished with processing the KDM database definition, it informs the KC. KC then notifies the user, via the LIL, that the database definition has been processed and that loading of the database records may begin. In the second task, KMS sends the KDL transactions to the KC. When KC receives the KDL transactions, it forwards them to KDS for execution. Upon completion, KDS sends the results in KDM form back to the KC. KC routes the results to the *kernel formatting system (KFS).* KFS reformats the results from KDM form to UDM form. KFS then displays the results in the correct UDM form via LIL.

The four modules, LIL, KMS, KC, and KFS, are collectively known as the *language interface.* Four similar modules are required for each of the other language interface of MLDS. For example, there are four sets of these modules where one set is for the hierarchical/DL/I language interface, one for the relational/SQL language interface, one for the network/CODASYL/DML language interface, and one for the entity-relationship/Daplex language interface. However, if the user writes the transaction in the native mode (i.e., in KDL), there is no need for an interface.

In our implementation of the network(CODASYL) language interface, we develop the code for the four modules. However, we do not integrate these modules with KDS as shown in Figure 1. The Laboratory of Database Systems Research at the Naval Postgraduate School is in the process of procuring new computer equipment for KDS. When the equipment is installed, KDS is to be ported over to the new equipment. MLDS software is then to be integrated with KDS. Although not a very difficult undertaking, it may be time-consuming.

C. THE KERNEL DATA MODEL AND LANGUAGE

The choice of a kernel data model and a kernel data language is the key decision in the development of a multi-lingual database system. The overriding question, when making such a choice, is whether the kernel data model and kernel data language is capable of supporting the required data-model transformations and data-language translations for the language interfaces.

The attribute-based data model proposed by Hsiao [Ref. 4], extended by Wong [Ref. 5], and studied by Rothnie [Ref. 6], along with the attribute-based

data language (ABDL), defined by Banerjee [Ref. 7], have been shown to be acceptable candidates for the kernel data model and kernel data language, respectively.

Why is the determination of a kernel data model and kernel data language so important for a MLDS? No matter how multi-lingual MLDS may be, if the underlying database system (i.e., KDS) is slow and inefficient, then the interfaces may be rendered useless and untimely. Hence, it is important that the kernel data model and kernel language be supported by a high-performance and great-capacity database system. Currently, only the attribute-based data model and the attribute-based data language are supported by such a system. This system is the multi-backend database system (MBDS) [Ref. 1].

D. THE MULTI-BACKEND DATABASE SYSTEM

The multi-backend database system (MBDS) has been designed to overcome the performance problems and upgrade issues related to the traditional approach of database system design. This goal is realized through the utilization of multiple backends connected in a parallel fashion. These backends have identical hardware, replicated software, and their own disk systems. In a multiple backend-configuration, there is a backend controller, which is responsible for supervising the execution of database transactions and for interfacing with the hosts and users. The backends perform the database operations with the database stored on the disk system of the backends. The controller and backends are connected by a communication bus. Users access the system through either the hosts or the controller directly (see Figure 2).

Performance gains are realized by increasing the number of backends. If the size of the database and the size of the responses to the transactions remain constant, then MBDS produces a reciprocal decrease in the response times for the user transactions when the number of backends is increased. On the other hand, if the number of backends is increased proportionally with the increase in databases and responses, then MBDS produces invariant response times for the same transactions. A more detailed discussion of MBDS is found in [Ref. 8].

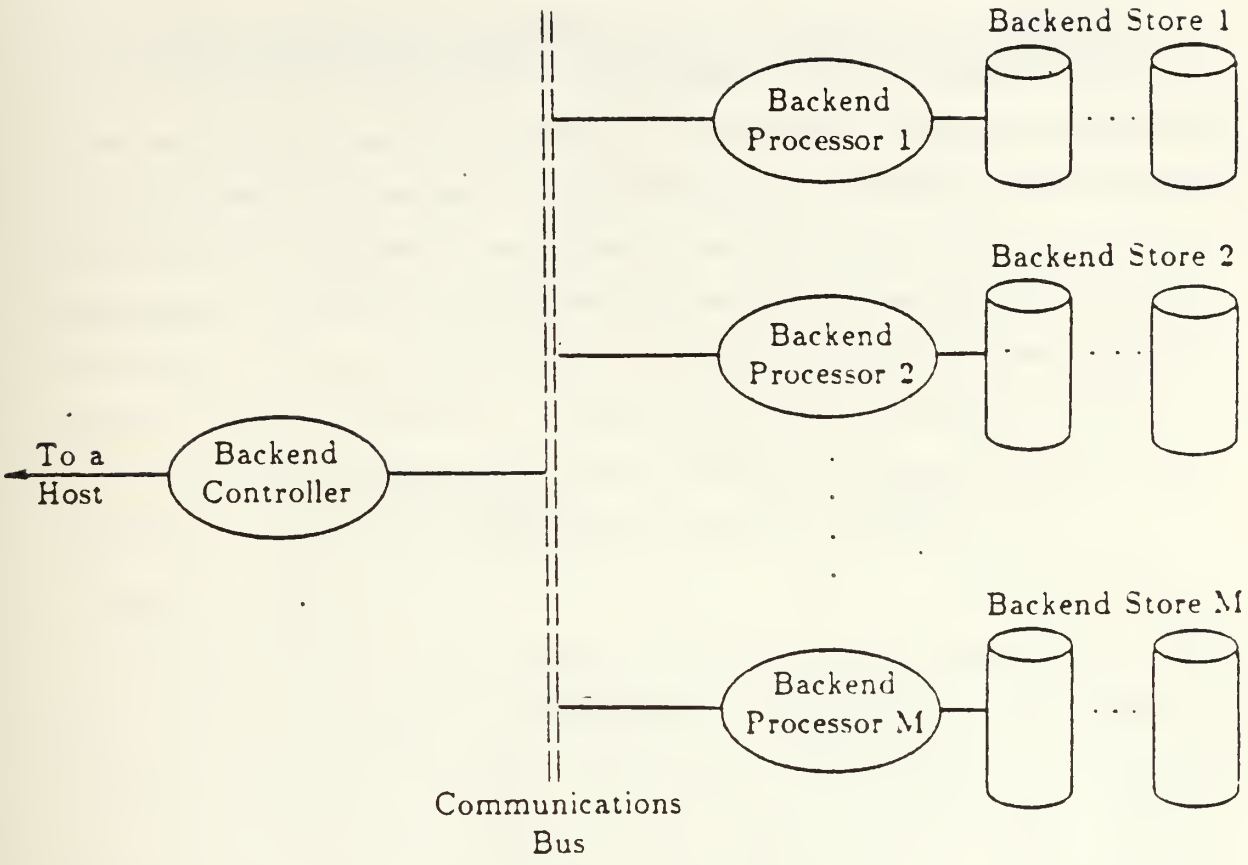


Figure 2. The Multi-Backend Database System.

E. THESIS OVERVIEW

The organization of our thesis is as follows: In Chapter II, we discuss the software engineering aspects of our implementation. This includes a discussion of our design approach, as well as a review of the global data structures used for the implementation. In Chapter III, we outline the functionality of the language interface layer. In Chapter IV, we articulate the processes constituting the kernel mapping system. In Chapter V, we provide an overview of the kernel controller. In Chapter VI, we describe the kernel formatting system. In Chapter VII, we conclude the thesis.

The detailed specifications of the interface modules (i.e., LIL, KMS, and KC) are given in Appendices A, B, and C, respectively. Appendix D is a users' manual for the system. The specifications of the source data language,

CODASYL-DML, and the target data language, ABDL, is found in [Ref. 9: pp. 389-446] and [Ref. 7], respectively.

Throughout this thesis, we provide examples of CODASYL-DML requests and their translated ABDL equivalents. All examples involving database operations presented in this thesis are based on the Suppliers_and_Parts sample database used by Date [Ref. 9: pp. 389-446]. The data structure diagram for this network is shown in Figure 3. There are supplier records (S), parts records (P), and shipments (SP) records. The sets of the database are suppliers-shipments (S-SP) and parts-shipments (P-SP).

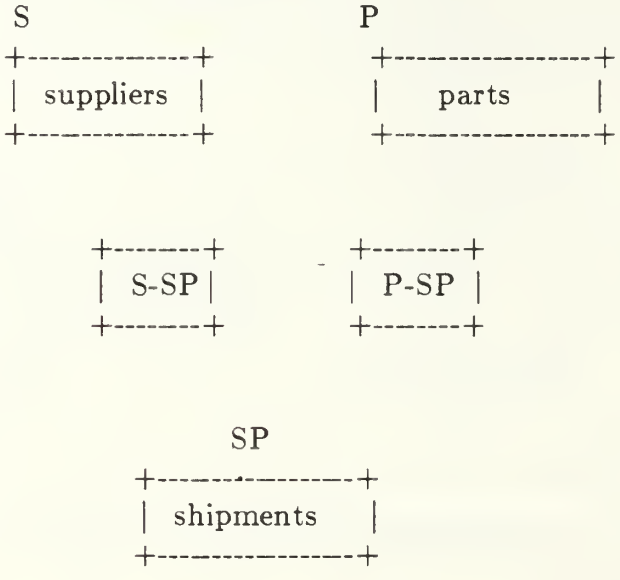


Figure 3. Data Structure Diagram of the Sample Suppliers-and-Parts Database.

II. SOFTWARE ENGINEERING OF A LANGUAGE INTERFACE

In this chapter, we discuss the various software engineering aspects of developing a language interface. First, we describe our design goals. Second, we outline the design approach that we took to implement the interface. Included in this section are discussions of our implementation strategy, our software development techniques, and salient characteristics of the language interface software. Then, we provide a critique of our implementation. Fourth, we describe the data structures used in the interface. And finally, we provide an organizational description of the next four chapters.

A. DESIGN GOALS

We are motivated to implement a CODASYL-DML interface for a MLDS using MBDS as the kernel database system, the attribute-based data model as the kernel data model, and ABDL as the kernel data language. It is important to note that we do not propose changes to the kernel database system or language. Instead, our implementation resides entirely in the host computer. All user transactions in CODASYL-DML are processed in the CODASYL-DML interface. MBDS continues to receive and process requests in the syntax and semantics of ABDL.

In addition, we intend to make our interface transparent to the user. For example, an employee in a corporate environment with previous experience with CODASYL-DML could log onto our system, issue a CODASYL-DML request and receive result data in a network format, i.e., a record. The employee requires no training in ABDL or MBDS procedures prior to utilizing the system.

B. AN APPROACH TO THE DESIGN

1. The Implementation Strategy

There are a number of different strategies we might have employed in the implementation of the CODASYL-DML language interface. For example,

there are the build-it-twice full-prototype approach, the level-by-level top-down approach, the incremental development approach, and the advancemanship approach [Ref. 10: pp. 41-46]. We have predicated our choice on minimizing the "software-crisis" as explained by Boehm [Ref. 10: pp. 14-31].

The strategy we have decided upon is the level-by-level top-down approach. Our choice is based on, first, a time constraint. The interface has to be developed within a specified time, specifically, by the time we graduate. And second, this approach lends itself to the natural evolution of the interface. The system is initially thought of as a "black box" (see Figure 1) that accepts CODASYL-DML transactions and then returns the appropriate results. The "black box" is then decomposed into its four modules (i.e., LIL, KMS, KC, and KFS). These modules, in turn, are further decomposed into the necessary functions and procedures to accomplish the appropriate tasks.

2. Techniques for Software Development

In order to achieve our design goals, it is important to employ effective software engineering techniques during all phases of the software development life-cycle. These phases, as defined by Ledthrum [Ref. 11: p. 27], are as follows:

(1) Requirements Specification - This phase involves stating the purpose of the software: what is to be done, not how it is to be done.

(2) Design - During this phase an algorithm is devised to carry out the specification produced in the previous phase. That is, how to implement the system which is specified during this phase.

(3) Coding - During this phase the design is translated into a programming language.

(4) Validation - During this phase it is ensured that the developed system functions as originally intended. That is, it is validated that the system actually performs what it is supposed to do.

The first phase of the life-cycle has already been performed. The research done by Demurjian and Hsiao [Ref. 1] has described the motivation, goals, and structure of the MLDS. The research conducted by Wortherly [Ref. 3] has extended this work to describe in detail the purpose of the CODASYL-DML interface. Hence, the requirements specification is derived from the above research.

We have developed the design of the system using the above specification. A Systems Specification Language (SSL) [Ref. 12] is used extensively during this phase. The SSL has permitted us to approach the design from a very high-level, abstract perspective by :

(1) enhancing communications among the program team members,

(2) reducing dependence on any one individual, and

(3) producing complete and accurate documentation of the design.

Furthermore, the SSL has allowed us to make an easy transition from the design phase to the coding phase.

We have used the C programming language [Ref. 13] to translate the design into executable code. Initially, we were not conversant in the language. However, our background in Pascal and the simple syntax of C have made it easy for us to learn. The greatest advantage of using C is the programming environment that it resides (i.e., the UNIX operating system). This environment has permitted us to partition the CODASYL-DML interface and then manage these parts in an effective and efficient manner. Perhaps, the only disadvantage with using C is the poor error diagnostics, having made debugging difficult. There is an on-line debugger available for use with C in UNIX for debugging. We have avoided this option and instead used conditional compilation and diagnostic print statements to aid in the debugging process. To validate our system we have used a traditional testing technique, i.e., path testing [Ref. 14]. We have tested those cases considered "normal". It is noteworthy to mention that testing, as we have done it, does not prove the system correct, but can only indicate the absence of problems with the cases that have been tested.

3. Characteristics of the Interface Software

In order for the CODASYL-DML interface to be successful, we have realized that it must be well designed and well structured. Hence, we are cognizant of certain characteristics that the interface must possess. Specifically, it must be simple. In other words, it must be easy to read and comprehend. The C code we have written has this characteristic. For instance, we often write the code with extra lines to avoid shorthand notations available in C.

These extra lines have made the difference between comprehensible code and cryptic notations.

The interface software also must be understandable. This must be true to the extent that a maintenance programmer, for example, can easily grasp the functionality of the interface and the relation between it and the other pieces of the system. Our software possesses this characteristic and does not have any hidden side-effects that could pose problems months or years from now. As a matter of fact, we have intentionally minimized the interaction between procedures to alleviate this problem.

The interface must also be maintainable. This is important in light of the fact that almost 70% of all of the software life-cycle costs are incurred after the software becomes operational, i.e., in the maintenance phase. There are software engineering techniques we employed that have given the CODASYL-DML interface this characteristic. For example, we require programmers to document changes to the interface code when the changes are made. Hence, maintenance programmers have current documentation at all times. The problem of trying to figure out the functionality of a program with dated documentation is alleviated. We also required the programmers to update their SSL specification as the code is being changed. Thus, the SSL specification consistently corresponds to the actual code. In addition, the data structures are designed to be general. Thus, it is an easy task to modify or rectify these structures to meet the demands of an evolving system.

The research conducted by Demurjian and Hsiao [Ref. 1] provides a high-level specification of the MLDS. The thesis written by Wortherly [Ref. 3] extends the above work and provides a more detailed specification of a CODASYL-DML language interface. This thesis outlines the actual implementation of a CODASYL-DML interface. The appendices provide the SSL design for this implementation.

A final characteristic that a CODASYL-DML interface should have is extensibility. A software product must be designed in a manner that permits the easy modification and addition of code. In this light, we have placed "stubs" in the correct locations of the KFS to permit the easy insertion of the code

needed to handle multiple horizontal screens of output. In addition, we have designed our data structures in such a manner that will permit subsequent programmers to easily extend them to handle not only multiple users, but also other language interfaces.

C. A CRITIQUE OF THE DESIGN

Our implementation of the CODASYL-DML interface possesses all of the elements of a successful software product. As noted previously, it is simple, understandable, maintainable, and extensible. Our constant employment of modern software engineering techniques have ensured its success.

However, there are two techniques that are especially worthy of critique. The first of these is the use of SSL. Initially, we have felt that the implementation language may also serve as the language to specify program algorithms. However, in doing so, we have stifled our creativity. This is because we are concentrating not only on what the algorithm does, but also on what the constructs (data structures) of the algorithm are. The use of SSL has permitted us to concentrate on the functionality of the algorithm without a heavy concentration on its particular constructs. This has allowed us to view the algorithm in a detached manner so that the most efficient implementation for the constructs can be used. Although we initially felt that the development of the program with SSL may be too time-consuming, our opinions changed when we realized the advantages of SSL, and the overall complexity of the CODASYL-DML language interface itself.

The way in which the data structures are designed is the other noteworthy software engineering technique. We have made extensive use of structures which are bound at compile time. We soon realize that in doing so, the computing resources (e.g., disk space) of the system are being depleted quite rapidly. Therefore, it is necessary for us to design the data structures in such a way that they can be managed in a dynamic fashion. Therefore, most of the data structures of the CODASYL-DML interface are linked lists. This design affords us the most convenient way to efficiently utilize the resources of the system. It is an easy task to use the C language's malloc (memory allocate) function to dynamically create the elements of a list as we need them.

In addition, the free command is useful in releasing these same elements for subsequent use.

D. THE DATA STRUCTURE

The CODASYL-DML language interface has been developed as a single user system that at some point will be updated to a multi-user system. Two different concepts of the data are used in the language interface : (1) Data shared by all users, and (2) Data specific to each user. The reader must realize that the data structures used in our interface and described below have been deliberately made generic. Hence, these same structures support not only our CODASYL-DML interface, but the other language interfaces as well i.e., DL/I, SQL, and Daplex.

1. Data Shared by All Users

The data structures that are shared by all users are the database schemas defined by the users thus far. In our case, these are network schemas, consisting of sets and attributes. These are not only shared by all users, but also shared by the four modules of the MLDS, i.e., LIL, KMS, KC, and KFS. Figure 4 depicts the first data structure used to maintain data. It is important to note that this structure is represented as a union. Hence, it is generic in the sense that a user can utilize this structure to support SQL, DL/I, CODASYL-DML, or Daplex needs. However, we concentrate only on the network(CODASYL) model. In this regard, the third field of this structure points to a record that contains information about a network(CODASYL) database. Figure 5 illustrates this record. The first field is just a character array

```
union dbid_node

    struct    rel_dbid_node    *rel;
    struct    hie_dbid_node    *hie;
    struct    net_dbid_node    *net;
    struct    ent_dbid_node    *ent;
```

Figure 4. The dbid node Data Structure.

```
struct net_dbid_node

    char            name[DBNLength + 1];
    int             num set;
    int             num rec;
    int             dbkey;
    struct nset_node    *first set;
    struct nset_node    *curr set;
    struct nrec_node    *first rec;
    struct nrec_node    *curr rec;
    struct net_dbid_node *next db;
```

Figure 5. The net dbid node Data Structure.

containing the name of the network database. The second and third fields contain an integer value representing the number of sets and the number of record types in the database. The fourth field also contains an integer value to give a different dbkey value to each record in the database. The fifth, sixth, seventh and ninth fields are pointers to other records containing information about each set and each record type in the database. Specifically, the fourth and sixth fields points to the first set and the first record type in the database while the fifth and seventh fields point to the current set and the current record type being accessed. The final field is just a pointer to the next network database.

The data structure nrec_node contains information about each record type in the database (see Figure 6). This structure is organized in much the same fashion that the net dbid node is organized. The first field of the data structure holds the name of the record type. The next field contains the number of attributes in this particular record type. The third and fourth fields point to other record types which contain data on the first and current attribute of this record type. And finally, the last field is a pointer to the next record type in this database.

The data structure nset node contains information about each set in the database (see Figure 7). The first field of the structure holds the name of the set. The second field contains the owner name of this set. The third field contains the

```
struct nrec node

    char                        name[RNLength + 1];
    int                         num attr;
    struct      nattr node      *first attr;
    struct      nattr node      *curr attr;
    struct      nrec node       *next rec;
```

Figure 6. The nrec node Data Structure.

```
struct nset node

    char                            name[SNLength + 1];
    char                            owner name[ONLength + 1];
    char                            member name[MNLength + 1];
    char                            insert mode[INLength + 1];
    char                            retent mode[RLength + 1];
    struct      set select node     select mode;
    struct      nrec node           *owner;
    struct      nrec node           *member;
    struct      nset node           *next set;
```

Figure 7. The nset node Data Structure.

member name of this set. The fourth and fifth fields serve as a flag to indicate the insertion and the retention mode. For instance, an insertion mode for a member record of a set can either be automatic or manual. Therefore,the characters "a", and "m" are used, respectively. The retention mode for a member record of a set can either be fixed, mandatory, or optional. Thus, the characters "f", "m", and "o" are used, respectively. The sixth field of this structure is a pointer to a data structure containing information about the set selection mode. The seventh field is a pointer to the owner record type of this set type. The eighth field is a pointer to the member record type of this set type. The final field is just a pointer to the next set type in the database.

The data structure set select node contains information about the set selection mode for each set in the database(See Figure 8). The first field serves as a flag to indicate the set selection mode. For instance, a set selection mode of a set type can either be by VALUE, by STRUCTURAL, by APPLICATION, or not specified. The characters "V", "S", "A", and "O" are used, respectively. If the set selection mode is by VALUE or by STRUCTURAL, the second field holds the item name of the specified record and the third field holds the name of the record. If the set selection mode is by STRUCTURAL, the fourth field holds the name of the second record , which is specified in the case of a by STRUCTURAL set selection criterion.

Figure 9 shows the organization of the final data structure used to support the definition of the network database schema. This structure contains information about the attributes of each CODASYL record type. The first field is an array, holding, the name of the attribute. The second field holds the level number of this attribute, and the third field serves as a flag to indicate the attribute type. For instance, an attribute can either be an integer, a floating point number, or a string. The characters "i", "f", and "s" are used, respectively to represent these types. The fourth field indicates the maximum length that a value of this attribute type may possibly have. For example, if this field is set to ten, and the type of this attribute is a string, then the maximum number of characters that a value of this attribute type may have is ten. The fifth field indicates the maximum length of the decimal portion of this attribute, if the type of this attribute is floating point. The sixth field is also a

```
struct  set select node

  char      select mode[SLength + 1];
  char      item name[ANLength + 1];
  char      record1 name[RNLength + 1];
  char      record2 name[RNLength + 1];
```

Figure 8. The set select node Data Structure.

```
struct nattr_node

    char                        name[ANLength + 1];
    char                        level_num[ALLength + 1];
    char                        type;
    int                         length1;
    int                         length2;
    int                         dup_flag;
    struct      nattr_node      *next_attr;
    struct      nattr_node      *child;
    struct      nattr_node      *parent;
```

Figure 9. The nattr_node Data Structure.

flag used to indicate whether or not this particular attribute can have duplicates. The seventh field is a pointer to the next attribute in this record. If the level number of an attribute is bigger than the previous level number, then the eighth field is used to reference a data structure that contains information on the child of the current attribute. If the level number of an attribute is less than the previous level number, then the ninth field is used to reference a data structure that contains information on the parent of the attribute. The reader may refer to Appendices A through C to examine how these data structures are used in SSL.

2. Data Specific to Each User

This category of data represents information needed to support each user's particular interface needs. The data structures used to accomplish this can be thought of as forming a hierarchy. At the root of this hierarchy is the data structure, user_info, that maintains information on all of the current users of a particular language interface (see figure 10). The user_info data structure holds the ID of the user, a union that describes a particular interface, and a pointer to the next user. The union field is of particular interest to us. As noted earlier, a union serves as a generic data structure. In this case, the union can hold the data for a user accessing either a CODASYL-DML language interface layer(LIL), a DL/I LIL, an SQL LIL, or a Daplex LIL. The li_info union is shown in Figure 11.

```
struct user info

    char                        uid[UIDLength + 1];
    union       li info         li type;
    struct      user info       *next user;
```

Figure 10. The user info Data Structure.

```
union li info

    struct   sql info   sql;
    struct   dli info   dli;
    struct   dml info   dml;
    struct   dap info   dap;
```

Figure 11. The li info Data Structure.

We are only interested in the data structures containing user information which relates to the CODASYL-DML language interface in this section. The structure used is referred to as dml info and is depicted in Figure 12. The first field of this structure, curr db info, is itself a data structure and contains currency information on the database being accessed by a user. The second field, file, is also a data structure. The file data structure contains the file descriptor and file identifier of a file of CODASYL-DML transactions, i.e., either queries or creates. The next field, dml tran, is also a data structure, and holds information that describes the CODASYL-DML transactions to be processed. This includes the number of requests to be processed, the first request to be processed, and the current request being processed. The fourth field of the dml info data structure, ddl files, is a pointer to a data structure which describes the descriptor and template files. These files contain information about the ABDL schema corresponding to the current network database being processed, i.e., the ABDL schema information for a newly defined network database.

The next field of the structure, operation is a flag that indicates the operation to be performed. This can be either the loading of a new database

```
struct dml info

    struct      curr db info    curr db;
    struct      file info       file;
    struct      tran info       dml tran;
    struct      ddl info        *ddl files;
    int                         operation;
    int                         answer;
    int                         error;
    union       kms info        kms data;
    union       kfs info        kfs data;
    union       kc info         kc data;
    struct      cur table       *cur table;
    int                         buff count;
```

Figure 12. The dml info Data Structure.

or the execution of a request against an existing database. The sixth field, answer, is used by the LIL to record the answer received through its interaction with the user of the interface. The remaining fields, kms data, kfs data, and kc data are unions that contain information required by the KMS, KFS, and KC. These are described in more detail in the next four chapters.The eleventh field points to records that implement the currency information table (CIT), as discussed by Meyer and MacDougal[Ref. 18]. The last field, buff count, is a counter variable used in KC to keep track of the result buffers.

E. THE ORGANIZATION OF THE NEXT FOUR CHAPTERS

The following four chapters are meant to provide the user with a more detailed analysis of the modules constituting the MLDS. Each chapter begins with an overview of what each particular module does and how it relates to the other modules. The actual processes performed by each module are then discussed. This includes a description of the actual data structures used by the modules. Each chapter concludes with a discussion of module shortcomings.

III. THE LANGUAGE INTERFACE LAYER (LIL)

LIL is the first module in the CODASYL-DML mapping process, and is used to control the order in which the other modules are called. The LIL allows the user to input transactions from either a file or the terminal. A transaction may take the form of either a database description of a new database, or a CODASYL-DML request against an existing database. A transaction may contain multiple requests. This allows a group of requests that perform a single task, such as a looping construct in CODASYL-DML, to be executed together as a single transaction. The mapping process takes place when LIL sends a single transaction to KMS. After the transaction has been received by KMS, KC is called to process the transaction. Control always returns to LIL, where the user may close the session by exiting to the operating system.

LIL is menu-driven. When the transactions are read from either a file or the terminal, they are stored in a data structure called net_req_info. If the transactions are schema definitions, they are sent to the KMS in sequential order. If the transactions are CODASYL-DML requests, the user is prompted by another menu to selectively choose an individual request to be processed. The menus provide an easy and efficient way for the user to view and select the methods of request processing desired. Each menu is tied to its predecessor, so that by exiting one menu the user is moved up the "menu tree". This allows the user to perform multiple tasks in one session.

A. THE LIL DATA STRUCTURES

LIL uses two data structures to store the user's transactions and control which transaction is to be sent to the KMS. It is important to note that these data structures are shared by both LIL and KMS.

The first data structure is named tran_info and is shown in Figure 13. The first field of this record, first_req, contains the address of the first transaction that has been read from a file or the terminal. The second field, curr_req, contains the address of the transaction currently being processed. LIL sets this

```
struct  tran_info
  {
   struct          net_req_info    *first_req;
   struct          net_req_info    *curr_req;
   int                             no_req;
  }
```

Figure 13. The tran_info Data Structure.

pointer to the transaction that the KMS is to process next, and then calls the KMS. The third field, no_req, contains the number of transactions currently in the transaction list. This number is used for loop control when printing the transaction list to the screen, or when searching the list for a transaction to be executed.

The second data structure used by LIL is named net_req_info. Each copy of this data structure represents a user transaction, and thus, is an element of the transaction list. The net_req_info data structure is shown in Figure 14.

The first field of this record, req, is a character string that contains the actual CODASYL-DML transaction. The second field, in_req, is a pointer to a list of character arrays that each contain a single line of one transaction. After all lines of a transaction have been read, the line list is concatenated to form the actual transaction, req. The third field of this record, req_len, contains the length of the transaction. It is used to allocate the correct and minimal amount of memory

```
struct  net_req_info
  {
   char                                 *req;
   struct          temp_str_info        *in_req;
   int                                  req_len;
   struct          net_req_info         *sub_req;
   struct          net_req_info         *next_req;
  }
```

Figure 14. The net_req_info Data Structure.

space for the transaction. If a transaction contains multiple requests, the fourth field, sub_req, points to the list of requests that make up the transaction. In this case, the field in_req is the first request of the transaction. The last field, next_req, is a pointer to the next transaction in the list of transactions.

B. FUNCTIONS AND PROCEDURES

LIL makes use of a number of functions and procedures in order to create the transaction list, pass elements of the list to the KMS, and maintain the database schemas. We do not describe each of these functions and procedures in detail. Rather, we provide a general description of the LIL processes.

1. Initialization

MLDS is designed to be able to accommodate multiple users, but is implemented to support only a single user. To facilitate the transition from a single-user system to a multiple-user system, each user possesses his own copy of a user data structure when entering the system. This user data structure stores all of the relevant data that the user may need during their session. All four modules of the language interface make use of this structure. The modules use many temporary storage variables, both to perform their individual tasks, and to maintain common data between modules. The transactions, in user data language form, and mapped kernel data language form, are also stored in each user data structure. It is easy to see that the user structure provides consolidated, centralized control for each user of the system. When a user logs onto the system, a user data structure is allocated and initialized. The user ID becomes the distinguishing feature to locate and identify different users. The user data structures for all users are stored in a linked-list. When new users enter the system, their user data structures are appended to the end of the list. In our current environment there is only a single element on the user list. In a future environment, when there are multiple users, we simply expand the user list as described above.

2. Creating the Transaction List

There are two operations the user may perform. A user may define a new database or process CODASYL-DML requests against an existing database. The first menu that is displayed prompts the user to select the

operation desired. Each operation represents a separate procedure to handle specific circumstances. The menu looks like the following:

```
Enter type of operation desired
    (l) - load a new database
    (p) - process old database
    (x) - return to the operating system
ACTION ----> _
```

For either choice (i.e., l or p), another menu is displayed to the user requesting the mode of input. This input may always come from a data file. If the operation selected from the previous menu had been "p", then the user may also input transactions interactively from the terminal. The generic menu looks like the following:

```
Enter mode of input desired
    (f) - read in a group of transactions from a file
    (t) - read in transactions from the terminal
    (x) - return to the previous menu
ACTION ----> _
```

Note that the "t" choice would be omitted if the operation selected from the previous menu had been to load a new database. Again, each mode of input selected corresponds to a different procedure to be performed. The transaction list is created by reading from the file or terminal, looking for an end-of-transaction marker or an end-of-file marker. These flags tell the system when one transaction has ended, and when the next transaction begins. When the list is being created, the pointers to access the list are initialized. These pointers, first_req and curr_req, have been described earlier in the data structure subsection. Both pointers are set to the first transaction read, in other words, the head of the transaction list.

3. Accessing the Transaction List

Since the transaction list stores both schema definitions and CODASYL-DML requests, two different access methods have to be employed to send the two types of transactions to KMS. We discuss the two methods

separately. In both cases, KMS accesses a single transaction from the transaction list. It does this by reading the transaction pointed to by the request pointer, curr_req, of the tran_info data structure (see Figure 13). Therefore, it is the job of the LIL to set this pointer to the appropriate transaction before calling KMS.

a. Sending schema definitions to KMS

When the user specifies the filename of a schema, (input from a file only) further user intervention is not required. To produce a new database, the transaction list of data definition statements is sent to KMS via a program loop. This loop traverses the transaction list, calling KMS for each data definition statement in the list.

b. Sending CODASYL-DML Requests to KMS

In this case, after the user has specified the mode of input, the user conducts an interactive session with the system. First, all CODASYL-DML requests are listed to the screen. As the requests are listed from the transaction list, a number is assigned to each transaction in ascending order, starting with the number one. The number appears on the screen to the left of the first line of each transaction. Note that each transaction may contain multiple requests. Next, an access menu is displayed which looks like the following:

```
Pick the number or letter of the action desired
    (num) - execute one of the preceding transactions
    (d)   - redisplay the list of transactions
    (x)   - return to the previous menu
ACTION ----> _
```

Since CODASYL-DML requests are independent items, the order in which they are processed does not matter. The user has the option of executing any number of CODASYL-DML requests. A loop causes the menu to be redisplayed after each CODASYL-DML request has been executed so that further choices may be made.

4. Calling KC

When KMS has completed its mapping process, the each transformed CODASTL-DML request have to be sent to KC to interface with the

kernel database system. Then, KC must update the currency information table (CIT) depending on the CODASYL-DML request. If, there are other CODASYL-DML requests on the same transaction, KMS continues its mapping process. Therefore, KC is immediately called, when its mapping process are completed for each CODASYL-DML request.

5. Wrapping-up

Before exiting the system, the user data structure described in Chapter II has to be deallocated. The memory occupied by the user data structure is freed and returned to the operating system. Since all of the user structures reside in a list, the exiting user's node has to be removed from the list.

IV. THE KERNEL MAPPING SYSTEM (KMS)

KMS is the second module in the CODASYL-DML mapping interface and is called from the language interface layer (LIL) when LIL receives CODASYL-DML requests from the user. The function of KMS is to: (1) parse the request to validate the user's CODASYL-DML syntax, and (2) translate, or map, the request to equivalent ABDL request(s). Once an appropriate ABDL request, or set of requests, has been formed, it is made available to the kernel controller (KC) which then prepares the request for execution by MBDS. KC is discussed in Chapter V.

A. AN OVERVIEW OF THE MAPPING PROCESS

From the description of KMS functions above we immediately see the requirement for a parser as a part of KMS. This parser validates the CODASYL-DML syntax of the input request. The parser grammar is the driving force behind the entire mapping system.

1. The KMS Parser / Translator

KMS parser has been constructed by utilizing Yet-Another-Compiler Compiler (YACC) [Ref. 15]. YACC is a program generator designed for syntactic processing of token input streams. Given a specification of the input language structure (a set of grammar rules), the user's code to be invoked when such structures are recognized, and a low-level input routine, YACC generates a program that syntactically recognizes the input language and allows invocation of the user's code throughout the recognition process. The class of specifications accepted is a very general one: LALR(1) grammars. It is important to note that the user's code mentioned above is our mapping code that is going to perform the CODASYL/DML-to-ABDL translation. As the low-level input routine, we utilize a Lexical Analyzer Generator (LEX) [Ref. 16]. LEX is a program generator designed for lexical processing of character input streams. Given a regular-expression description of the input strings, LEX generates a program that

partitions the input stream into tokens and communicates these tokens to the parser.

The parser produced by YACC consists of a finite-state automaton with a stack. It performs a top-down parse, with left-to-right scan and one token look-ahead. Control of the parser begins initially with the highest-level grammar rule. Control descends through the grammar hierarchy, calling lower and lower-level grammar rules which search for appropriate tokens in the input. As the appropriate tokens are recognized, some portions of the mapping code may be invoked directly. In other cases, these tokens are propagated back up the grammar hierarchy until a higher-level rule has been satisfied, at which time further translation is accomplished. When all of the necessary lower-level grammar rules have been satisfied and control has ascended to the highest-level rule, the parsing and translation processes are complete. In Section B, we give an illustrative example of these processes.

2. The KMS Data Structures

KMS utilizes, different kinds of structures for different kinds of requests. It, naturally, requires access to the CODASYL-DML input request structure discussed in Chapter II, the dml_tran structure.

CIT has been described in Chapter 2. This structure carries all of the currency information for a particular run unit, and is vital to the proper translation and execution of CODASYL statements. LIL of the interface should initialize CIT. The KMS should have read access to CIT at all times, while any updates of the CIT should be done by KC only.

The following data structures will be needed in KMS, and each is directly associated with a particular CODASYL statement. The first field of the net_kms_info structure, shown in Figure 15, is a pointer to the data structure that contains duplication information accumulated by the KMS during the grammar-driven parse. This data structure contains the name of an attribute that has DUPLICATES NOT ALLOWED specified in the schema definition, and a pointer to the next attribute with the same specification (see Figure 16). We use this list when setting the non_duplicate flags in the attribute nodes.

```
struct net_kms_info
 {
  struct        ndup_node     *ndup_list;
  struct        move_list     *move_list;
  struct        select_list   *select_list;
  struct        connect_list  *connect_list;
  struct        abdl_req      *abdl;
  struct        erase_abdl    *erase;
  struct        find_abdl     *find;
  struct        store_abdl    *store;
  struct        get_node      *get;
  struct        find_abdl     *cur_find;
 }
```

Figure 15. The net_kms_info Data Structure.

```
struct ndup_node
 {
  char                  name[ANLength + 1];
  struct        ndup_node   *next;
 }
```

Figure 16. The ndup_node Data Structure.

The second field of the net_kms_info structure is a pointer to the head of the move_list structure (see Figure 17). The move_list simulates the MOVE statements used as assignment statements by the host language COBOL in other CODASYL implementations. In our implementation, we create the move_list structure to keep track of these assignment statements, and to validate the execution of other CODASYL-DML statements. The first field and second field of the structure point to the record template structure (see Figure 17). This structure keeps track of the name of the record type in the move statements. The third field of the move_list points to the data item structure (see Figure 17). Each data item contains the attribute name, attribute type, and value information corresponding to the item that is the object of the MOVE statement. It should be noted that the value field in the data item record is a pointer to a variable-length character string. Although attribute names have a constant maximum-length constraint, the length of attribute values in the database is

limited only by the constraint placed on them by the user in the original database definition, and as such, they may be of varying lengths.

The third field of the net_kms_info structure is a pointer to the select_list data structure (see Figure 18). This data structure contains attribute names to be used to retrieve records from the database. The fourth field of the net_kms_info structure is a pointer to the connect_list data structure (see Figure 19). This data structure contains set type names to be connected or disconnected.

```
struct  move_list
  {
    struct              record_template    *first_rec;
    struct              record_template    *cur_rec;
    struct              data_item          *cur_item;
  }

struct  record_template
  {
    char                                   name[RNLength + 1];
    struct              record_template    *next_record;
    struct              data_item          *item_list;
  }

struct  data_item
  {
    char                                   name[ANLength + 1];
    char                                   *value;
    char                                   type;
    struct              data_item          *next_item;
  }
```

Figure 17. The Move_list Data Structures.

```
struct  select_list
  {
    char                                   *item_name;
    struct              select_list        *next_item;
  }
```

Figure 18. The select_list Data Structure.

```
struct   connect_list
 {
  char                                  set_type[SNLength + 1];
  struct              connect_list      *next_set_type;
 }
```

Figure 19. The connect_list Data Structure.

The fifth field of the net_kms_info structure is a pointer to the abdl_req data structure (see Figure 20). The first field of this structure is a pointer to the actual ABDL request generated by the KMS, for DISCONNECT, CONNECT and MODIFY CODASYL-DML requests. The second field, operation, defines the type of request to be executed. The last field is a pointer to other records of the same type.

The sixth field of the net_kms_info structure is a pointer to the erase_abdl data structure (see Figure 22). This structure contains information about the ERASE CODASYL-DML request. The fist field of this data structure defines the type of request to be executed (i.e., ERASE ALL or ERASE specific record_type). The second and third fields are pointers to the actual ABDL request generated by the KMS. The fourth field is a pointer to the member_erase data structure. We use this field when the CODASYL-DML request is ERASE ALL. If the given record_type is an owner of a non_empty set, then we delete all of the members of the set owned by this record_type (see

```
struct    abdl_req
{
 char                                   *abdl;
 int                                    operation;
 struct              abdl_req           *next_abdl;
}
```

Figure 20. The abdl_req Data Structure.

Figure 21). The last field, result_file, is used in KC to accumulate results obtained from MBDS when executing the ABDL requests.

The seventh field of the net_kms_info structure is a pointer to the find_abdl data structure (see Figure 23). The first field of this data structure, set_type, contains the set name of the translated request. The second field, rec_type, contains the record name of the translated request. The third field, abdl, is a pointer to the actual ABDL request generated by KMS. The fourth field, num_attr, contains the number of attributes in this particular record type of the translated request. The fifth field, operation, defines the type of FIND request to be executed (i.e., FIND ANY, etc.). The sixth field, dont update, indicates that there is no need to update the CIT table for the request being

```
struct   member_erase
{
 char                                set_name[SNlength + 1];
 char                                *abdl;
 char                                *delete;
 char                                *template;
 struct                net_file_info *result_file;
 struct                member_erase  *eraseall,
                                     *next;
}
```

Figure 21. The member_erase Data Structure.

```
struct   erase_abdl
{
 int                                 operation;
 char                                *abdl;
 char                                *retrieve;
 struct                member_erase  *member;
 struct                net_file_info *result_file;
}
```

Figure 22. The erase_abdl Data Structure.

```
struct  symbolic_info
{
char                                name[ANLength + 1];
struct              symbolic_info   *next;
}

struct  suppres_list
{
 char                               set_type[SNLength + 1];
 struct             suppres_list    *next;
}

struct  set_list
{
 char                               set_name[SNLength + 1];
 char                               owner_name[RNLength + 1];
 char                               *dbkey;
 struct             set_list        *next;
}

struct   find_abdl
{
 char                               set_type[SNLength + 1];
 char                               rec_type[RNLength + 1];
 char                               *abdl;
 int                                num_attr;
 int                                operation;
 int                                dont_update;
 struct             set_list        *set_list;
 struct             suppres_list    *suppres;
 struct             symbolic_info   *tgt_list;
 struct             find_abdl       *next;
 struct             net_file_info   *result_file;
}
```

Figure 23. The find_abdl Data Structures.

processed. The seventh field, set_list, is a pointer to the set_list data structure. This structure is a list of set names. We use this field, when we update CIT table. The eighth field, suppres, is used by the KC to update the CIT table. The ninth field, tgt_list is a pointer to the symbolic_info data structure. This

structure is a list of attribute names (the target list). The last field, result_file is used in KC to accumulate results obtained from MBDS when executing the ABDL requests.

The eighth field of the net_kms_info structure is a pointer to the store_abdl data structure (see Figure 24). This structure contains information about the STORE CODASYL-DML request. The first field, rec_name, contains the record name of the translated request. The second field, ret_abdl is a pointer to a RETRIEVE request. This request is generated by the KMS in order to determine the existence of duplicate values for data items declared to have DUPLICATES NOT ALLOWED in the database schema. The third field, ins_abdl, is a pointer to the INSERT request which will actually cause the record to be placed into the database. The fourth field, template1, is used as working

```
struct   ret2_node
{
 char                          set_name[SNLength + 1];
 char                          owner[RNLength + 1];
 char                          select_mode[SLength + 1];
 char                          insert_mode[INLength + 1];
 char                          *abdl;
 int                           flag;
 struct          net_file_info *result_file;
 struct          ret2_node     *next;
}

struct   store_abdl
{
 char                          rec_name[RNLength + 1];
 char                          *ret_abdl;
 char                          *ins_abdl;
 char                          *template1;
 int                           dont_update;
 struct          ret2_node     *ret2_abdl;
 struct          suppres_list  *suppres;
 struct          net_file_info *result_file;
}
```

Figure 24. The store_abdl Data Structures.

space by the KC. The fifth field, dont_update, indicates that there is no need to update the CIT table for the request being processed. The sixth field, ret2_abdl, is a pointer to a RETRIEVE request which returns the owner database key value of the proper set occurrence for the new record (see Figure 24). The first field of the ret2_node structure, set_name, contains the set name of the record to be inserted. The second field, owner, contains the name of the set type of the record to be inserted. The third field, select_mode defines the set selection criteria for the record being stored. The fourth field, insert_mode defines the set insertion mode for the record being stored. The fifth field, abdl, is a pointer to a RETRIEVE request which returns the owner database key value of all the set occurrences to which this new record belongs. The sixth field, flag, is used in KC to build the INSERT request. If we do not insert the new record into a set type. Then, we set this flag. The seventh field, result_file, is used in the KC to accumulate results obtained from MBDS when executing the ABDL requests. The seventh field of the store_abdl data structure, suppres, is used by KC to update the CIT table. The last field, result_file, is used in KC to accumulate results obtained from MBDS when executing the ABDL requests.

The ninth field of the net_kms_info structure is a pointer to the get_node data structure (see Figure 25). This structure contains information about the GET CODASYL-DML request. The first field, type, contains the record name in question. The second field, operation, identifies the type of GET format being used (i.e., GET record_type or GET item_list IN record_type). The third field,

```
struct   get_node
{
 char                          type[RNLength + 1];
 int                           operation;
 struct          select_list   *tgt_list;
 struct          get_node      *next;
}
```

Figure 25. The get_node Data Structure.

tgt_list, is a pointer to the select_list data structure. It includes a list of items to be returned. If the format is GET record_type, this field would be NULL, and KC would return all attributes of the record. The same is true for the simple GET format. The last field is a pointer to other records of the same type. Thus, these records are connected in a linearly_linked list.

The tenth field of the net_kms_info structure is a pointer to the last FIND request. We use this field to display correct result buffer when the user issue GET request.

B. FACILITIES PROVIDED BY THE IMPLEMENTATION

In this section, we discuss those CODASYL-DML facilities that are provided in our implementation of the network (CODASYL) interface. We do not discuss the CODASYL/DML-to-ABDL translation in detail. This subject is discussed by Wortherly [Ref. 3]. Rather, we provide an overview of the salient features of KMS, accompanied by one illustrative example of the parsing and translation process. User-issued requests may take two forms, schema definition statements, or CODASYL-DML database manipulations. Appendix B contains the design of our implementation, written in a system specification language (SSL).

1. Database Definitions

When the user informs the LIL that the user wishes to create a new database, the job of KMS is to build a schema that corresponds to the schema definition statements input by the user. The LIL initially allocates a new database identification node (net_dbid_node shown in Figure 5) with the name of the new database, as input by the user. The LIL then sends the KMS a complete schema definition, which has the form :

```
SCHEMA NAME IS database_name ;
RECORD NAME IS record_type ;
  DUPLICATES ARE NOT ALLOWED FOR attr_name ;
  01 attr_1 ; CHARACTER length .
    attr_2 ; FIXED length .
      .
      .
      .
RECORD NAME IS record_type ;
  DUPLICATES ARE NOT ALLOWED FOR attr_name, attr_name ;
```

```
    attr_1 ; CHARACTER length .
      .
      .
SET NAME IS set_type ;
  OWNER IS record_type ;
  MEMBER IS record_type ;
    INSERTION IS insertion_mode
    RETENTION IS retention_mode ;
    set_selection_mode ;
SET NAME IS set_type ;
      .
      .
```

The sequence of statements in the schema definition is significant. First, all record declarations have to appear, followed by all declarations for each record statement, an additional record node (nrec_node shown in Figure 6) is added to the database schema under construction. For each subsequent attribute statement, an additional attribute node (nattr_node shown in Figure 9) is added to the schema for the current record under construction. Then, for each set statement, an additional set node (nset_node shown in Figure 7) is added to the database schema under construction. The database identification node (net_dbid_node shown in Figure 5) holds the number of records and the number of sets in the schema, the database name, and the initial value of dbkey. Each record node holds the number of attributes in that record, and the record name. Each attribute node holds the attribute name, level number, length, type, and non_duplicate flag value. Each set node holds the set name, the owner's name, and the member's name, the insertion mode, the set selection mode, and the retention mode.

When KMS has parsed all of the statements included in the schema definition, the result is a completed database schema, as shown in Figure 26. Not shown in Figure 26 is the list of attribute nodes that is connected to each record node. The network (CODASYL) database schema, when completed, serves two purposes. First, when creating a new database, it facilitates the construction of the MBDS template and descriptor files. Secondly, when processing requests against an existing database, it allows validitiy checks of the records, sets, and attribute names used. It also serves as a source of information for type_checking.

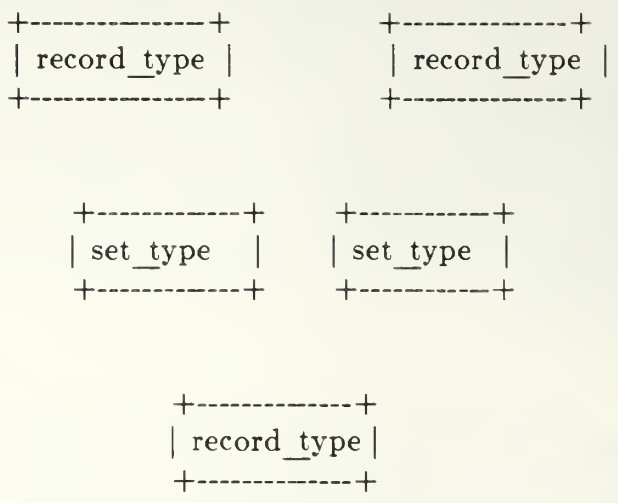


Figure 26. The Network Database Schema.

2. Database Manipulations

When the user wishes the LIL to process requests against an existing database, the task of KMS is to map the user's CODASYL-DML request to equivalent ABDL requests.

a. The Mapping Processes: An Example

In this subsection we present an illustrative example of the KMS mapping process (i.e., parsing and translation) for a simple CODASYL-DML FIND ANY call. We begin by showing the grammar for the dml portion of KMS. We then step through the grammar and demonstrate appropriate portions of our design in the system specification language (SSL). We only show those portions of the design that are relevant to the example, i.e., those that would actually be executed. The entire KMS design is shown in Appendix C. The relevant grammar is shown in Figure 27. The source CODASYL-DML call to be utilized for our example is the following:

```
MOVE Cleveland TO CITY IN SA
FIND ANY SA USING CITY IN SA
```

(Note: The MOVE statement is an assignment statement found in the host COBOL language.) The ABDL request generated in response to such a CODASYL-DML call is as follows:

```
[ RETRIEVE ((TEMPLATE = SA) and (CITY = Cleveland))
        (SNO, SNAME, STATUS, CITY, DBKEY) BY DBKEY ]
```

To begin our discussion, let us first synchronize the reader. At the beginning of the mapping process, the parse descends the grammar hierarchy searching for appropriate tokens in the input that may satisfy one of the grammar rules. Therefore, the parser descends through the ddl_statement rules (schema definition statements). After finding no matching tokens for these rules, the parser eventually descends to the dml rule (data manipulation language).

When the dml rule is first called, it immediately calls the dml_statement rule, then starts to search for appropriate tokens in the input that satisfy one of its sub_rules. In our example, the move rule is called. For the sake of brevity in the example, we will not go through the mapping process for the MOVE statement. We need only be aware that the new value for the attribute CITY in the record template for record type SA, has been set to the value Cleveland, by the previous parse/translation. Now, we may proceed with the mapping of the FIND ANY statement.

When the find rule is called, the FIND token is recognized satisfying the first portion of the rule. Control now goes to the record_selection_expr rule. This rule then searches for tokens in the input that satisfy one of its sub_rules. In our example, the ANY record_type portion of this rule is satisfied. The record_type rule recognizes the token, SA, via the terminal, IDENTIFIER. At this point, we need to perform some translation. The following SSL is invoked before the remaining portion of the rule is satisfied.

```
statement: ddl_statement
        | dml

dml: dml_statement
   | dml dml_statement

dml_statement: set_flag
             | move
             | get
             | find
             | store
             | connect
             | disconnect
             | erase
             | modify
             | perform_loop
             | if_then

find: FIND record_selection_expr  curr_suppression

record_selection_expr: ANY record_type USING item_list
                     IN record_type

curr_suppression: LSQUARE supp_expr RSQUARE
                | empty

item_list: item_name
         | item_list COMMA item_name

record_type: IDENTIFIER

item_name: IDENTIFIER
```

Figure 27. The KMS dml_statement Grammar.

```
record_selection_expr: ANY record_type
      {
       if ('record_type' record_template node is
           not on move_list)
         perform error(1)
         return
       else
        alloc and init new 'find' node
```

```
    find_type = ANY in find node
    copy record_type to find node
    alloc and init new abdl_str
    alloc and init new tgt_list
    /* begin forming a RETRIEVE request */
    copy "[ RETRIEVE (( TEMPLATE = 'record_type')"
      to abdl_str
   end_if
   select_list = NULL
  }
USING item_list IN record_type
      | CURRENT record_type WITHIN set_type
                .
                :
                :
```

We first check to see if the record_type is on the move_list. If it is not, the system gives an error message and exits the parser. If it is on the move_list, we allocate and initialize a new find node. At the same time, we initialize an abdl string to be used for forming the ABDL request. The target list is also allocated and initialized at this point. Next, we copy "[RETRIEVE ((TEMPLATE = 'record_type')" to the abdl_str. Then, we free the select_list. The select list holds the attribute names from the item_list rule.

The next token encountered is the USING token. It matches USING in the FIND ANY rule. So, the item_list rule below is called. This rule recognizes the token, CITY, in the CODASYL-DML call and creates the select_list.

```
item_list: item_name
        {
         put the first item_name on select_list
        }
       | item_list COMMA item_name
        {
         put successive item names on select_list
        }
```

The IN token is recognized next and satisfied. Control is then passed to the record_type rule. This rule recognizes the token, SA, and the parsing process is complete. We must now perform more translation in order to complete the record_selection_expr rule as indicated below.

```
record_selection_expr: ANY record_type
                       {
                        :
                       }
      USING item_list IN record_type
      {
       if ('record_type' is same as previous
            'record_type')
         if (any data item on select_list is not
            defined for 'record_type')
            perform error(2)
            return
         end_if
         else
          create tgt_list item for all attributes
            of 'record_type' record
          for (each data item on select_list)
            if ('data_item' not on move_list )
              perform error(1)
              return
            end_if
            else
              get 'item_value' from move_list
              concat "and ('data_item' = 'item_value')"
                to abdl_str
            end_else
          end_for
          concat ")('tgt_list')" to abdl_str
          checkmember()
          concat "DBKEY) BY DBKEY]" to abdl_str
          connect abdl_str to find node
         end_else
       end_if
       else
        perform error(6)
        return
       end_else
      }
     | CURRENT record_type WITHIN set_type
                        :
                        :
```

First, we check to see if the last record_type is the same as the previous record_type. If it is not, the system gives an error message and exits the parser. If it is correct, we must check the select_list for any attribute names that

are not defined for the record_type. If undefined names are on the select_list, the system gives an error message and exits the parser. If there are no undefined names we must create a tgt_list for all of the attributes of the record_type. Then. we must check each attribute name on the select_list for inclusion on the move_list, because we need the value of the attribute in order to issue the RETRIEVE request. If the attribute is not on the move_list, the system gives an error message and exit the parser. If the attribute is on the move_list, we must concatenate each attribute name on the select_list, and it's value from the move_list to the abdl_str. Next, we concatenate ")('tgt_list') " to the abdl_str. (Note: We created the tgt_list earlier, thus, we simply concatenate that list to the abdl_str.) Next, we check to determine if the record in question is a member of any set. If the record does belong to one or more sets, we concatenate the MEMset_type attribute for those sets to the target list of the abdl_str being processed. We use the MEMset_type values to update the CIT table properly. Next, we concatenate "DBKEY) BY DBKEY]" to the abdl_str.

Now, the record_selection_expr rule is completed, and control returns to the curr_suppression rule. The empty portion of the curr_suppression rule is matched, satisfying the curr_suppression rule. Now, the dml_statement and dml rules are fully satisfied, and control returns to the start statement. The parsing and translation process for our example is now completed and the find_node is passed to the KC for execution.

V. THE KERNEL CONTROLLER

The Kernel Controller (KC) is the third module in the MLDS CODASYL language interface. It is called by the language interface layer (LIL) when a new database is being loaded, and is called by the kernel mapping system (KMS) when an existing database is being manipulated. KC is the module which performs the task of controlling the submission of ABDL transactions to the multi-backend database system (MBDS) for processing.

KC must perform the following functions: (1) submit transactions to the MBDS, (2) receive and store results of transactions, (3) update the currency information table, and (4) cause the proper data to be returned to the user.

The procedures that make up the interface to the KDS (i.e., MBDS) are contained in the test interface (TI) of MBDS. To fully integrate the KC with the KDS, the KC calls procedures which are defined in the TI. Due to upcoming hardware changes in MBDS, we decided not to test the KC on-line with the TI. Our solution to this problem has been to design the system exactly as if it were interfacing with the TI. However, for each call to a TI procedure, we have created a software stub that performs the same functions as the actual TI procedure. The reader should realize that all interactions with the TI procedures described in the KC are actually made with these software stubs, rather than with the on-line TI procedures.

In this chapter we discuss the processes performed by the KC. This discussion is in two parts. First, we examine the data structures relevant to the KC, followed by an examination of the functions and procedures found in the KC. Appendix C contains the design of our KC implementation, written in a system specification language (SSL).

A. THE KC DATA STRUCTURES

In this section, we review some of the data structures discussed in Chapter II, focusing on those structures that are accessed and used by KC. One data structure used by KC is the dml_info record shown in Figure 28. KC makes use

of only four fields in this record. The first field, operation, defines what action is going to be taken by KC.

The second field, buff_count, is an integer used to maintain control of the file buffers associated with the results of each RETRIEVE request.

The third field, kms_data, is a pointer to the kms_info union data structure. This structure points to the net_kms_info data structure, which allows us to execute proper ABDL request(s).

The fourth field, cur_table, is a pointer to the cur_table data structure (see Figure 29), which contains currency information for the database in use. The first field of the cur_table data structure points to the run_unit data structure. This data structure contains information about the current of run_unit. The term "run_unit" means the most recently accessed record of any type whatsoever. The first field of the run_unit data structure, holds the name of the current record of the run_unit. The second field, dbkey, holds the database key value of the current record of the run_unit. We do not use any information relating to the current record of the record type in our KMS implementation. Thus, we do not

```
struct dml_info
  {
   struct          curr_db_info    curr_db;
   struct          file_info       file;
   struct          tran_info       dml_tran;
   struct          ddl_info        *ddl_files;
   int                             operation;
   int                             answer;
   int                             error;
   int                             buff_count;
   union           kms_info        kms_data;
   union           kfs_info        kfs_data;
   union           kc_info         kc_data;
   struct          cur_table       *cur_table;
  }
```

Figure 28. The dml_info Data Structure.

```
struct  run_unit
  {
   char                          rec_type[RNLength + 1];
   int                           dbkey;
  }

struct  cur_set
  {
   char                          set_name[SNLength + 1];
   char                          type[RNLength + 1];
   int                           dbkey;
   char                          member[RNLength + 1];
   char                          owner[RNLength + 1];
   int                           owner_dbkey;
   struct          cur_set       *next_set;
  }

struct   cur_table
  {
   struct          run_unit      *run;
   struct          cur_set       *set_type;
   struct          cur_set       *cur_set;
  }
```

Figure 29. The cur_table Data Structure.

include the current of record type in our implementation of the currency information table. The second and third fields of the cur_table data structure, point to the cur_set data structure. This data structure contains information about the current of set type. The first field of this data structure contains the name of the set type. Note that sometimes the current of set type is going to be an owner, and sometimes it is going to be a member. The second field of the cur_set data structure contains this information. The third field, dbkey, holds the database key value of the current of set type. The fourth and fifth fields hold the name of the member and owner record_types of the set type. The sixth field, owner_dbkey, holds the database key value of the owner record of the set type. We create the cur_set data structures dynamically. If the set type being processed is not on the currency table, then we create a new cur_set data

structure to hold information about this set type. The last field of the cur_set data structure is a pointer to other data structures of the same type that connect the data structures in a linearly_linked list.

The net_file_info is used by KC to store information about the file buffers containing the results obtained for each RETRIEVE request (see Figure 30). The first field, buff, contains the file name and file id. This information is required so that the appropriate files may be written to and read from, as necessary. The second field, count, is simply an integer representing the number of results in the file buffer. The next field, buff_loc, indicates the KC's location in the file buffer. For instance, after the first value is pulled from a file buffer, this field indicates that the KC's position is now at the beginning of the second result. The fourth field, status, serves as a flag so that a file buffer is opened under the correct status. The fifth field, max_chars, defines the maximum length of response in the result buffer. The sixth field is a pointer to a character string that holds the last result value pulled from the file buffer. This value may be used in the building of subsequent requests, or it may be used to update the CIT

```
struct  file info
  {
   char                          fname[FNLength + 1];
   FILE                          *fid;
  }

struct   net_file_info
  {
   struct              file_info   buff;
   int                             count;
   int                             buff_loc;
   int                             status;
   int                             max_chars;
   char                            *curr_buff_val;
   char                            *tem_str;
  }
```

Figure 30. The net_file_info Data Structure.

table. The last field is a pointer to a character string that holds the response record. This field is used by KFS to display the result to the user.

B. FUNCTIONS AND PROCEDURES

The KC makes use of a number of different functions and procedures to manage the transmission of the translated CODASYL-DML requests (i.e., ABDL requests) to the KDS. Not all of these functions and procedures are discussed in detail. Instead, we provide the reader with an overview of how the KC controls the submission of the ABDL requests to MBDS.

1. The Kernel Controller

The kernel_controller procedure is called by LIL when a new database is being loaded, and is called by KMS when an existing database is being manipulated. This procedure provides the master control over all other procedures used in KC. This procedure is a case statement that calls different procedures based upon the type of ABDL transactions being processed. If a new database is being created, the load_tables procedure is called. If the transaction is of any other type, then the appropriate procedure for processing that transaction is called. If the transaction is none of the above, there is an error, and an error message is generated with control returned to LIL.

2. Creating a New Database

The creation of a new database is the least difficult transaction that the KC handles. The load_tables procedure is called, which performs two functions. First, the test interface (TI) dbl_template procedure is called. This procedure is used to load the database-template file created by the KMS. Next, the TI dbl_dir_tbls procedure is called. This procedure loads the database-descriptor file. These two files represent the attribute-based metadata that is loaded into the KDS, i.e., MBDS. After execution of these two procedures, control returns to the LIL.

3. The FIND Requests

The find_requests_handler procedure is called by the Kernel_Controller procedure to handle FIND requests. The find_requests_handler procedure is a large case statement. The find_requests_ handler procedure takes action depending on the FIND request being processed. If the FIND request is either

the FIND ANY, FIND FIRST, FIND LAST, FIND WITHIN or FIND OWNER , find_requests_handler takes the same action. It executes the RETRIEVE request associated with the statement, and calls the Find_update procedure. If the FIND request is either a FIND NEXT, FIND PRIOR or FIND DUPLICATE, the find_requests_handler procedure must first find the correct find_abdl data structure and determine the correct buffer location in the result file of this data structure. The procedure then calls the Find_update procedure. If the FIND request is a FIND CURRENT, then find_requests_handler simply calls the Find_update procedure.

The Find_update procedure is called by the find_requests_handler, if the user does not use SUPPRESS UPDATE mode. The main goal of this procedure is to set up currency information depending on the type of find request.

The following examples illustrate the logic used in this procedure. Suppose the following CODASYL-DML request is issued by the user:

```
MOVE SS5 TO SNO IN SP
FIND ANY SP USING SNO IN SP
```

KMS translates this request into the following ABDL RETRIEVE request:

```
[ RETRIEVE (( TEMPLATE = SP ) and ( SNO = SS5 ))
  (SNO, PNO, QTY, MEMSSP, MEMPSP, DBKEY) BY DBKEY ]
```

The Kernel_Controller procedure is then called by KMS to execute this request and update the CIT table. The Kernel_Controller procedure thhen calls the find_requests_handler procedure. This procedure provides the master control over all FIND requests. In our example, case AnyFin statement is satisfiyied. Since the RETRIEVE request is complete, it may be immediately forwarded to KDS for execution. This is accomplished by calling dml_execute. This procedure uses two TI procedures and the dml_check_requests_left procedure. In general, dml_execute sends the ABDL request to KDS and waits for the last response to be returned. Results for a given request are placed in a unique file buffer associated with each request data structure. The file_results procedure controls this process.

After the last response is returned, the Find_update procedure takes control. The action taken is dependent upon the particular case satisfied in the procedure. In our example, case AnyFin statement is satisfied. The SP record_type is a member of both SSP and PSP set types. If the user does not specify any currency suppression, we update the currency of all set types to which this record type belongs, and the currency of the run_unit. If the set types are not on the CIT table, we create new cur_set data structures to hold the necessary information, if the user specifies set types for currency suppression, we only update the currency of these particular set types and the currency of the run_unit. If the user specifies SUPPRESS UPDATE, then we do not update the CIT table.

Finally, control returns to KMS via the Kernel_Controller procedure and the find_requests_handler procedure. If there is another request in the same transaction, KMS continues the mapping process. If not, control returns to LIL, and we can pick another transaction, or return to any of the other MENUs in LIL.

4. The Modify, Connect, and Disconnect Requests

If the request is Modify, Connect, or Disconnect, the request_handler procedure is called by the Kernel_Controller procedure. The first thing done by the request_handler procedure is to execute all ABDL UPDATE requests created by KMS. If the request is Connect or Disconnect, the request_handler procedure takes appropriate action to update the CIT table. We use the connect_list data structure (see Figure 19) to update set type(s) correctly.

5. The Move Request

There is no action taken by the Kernel_Controller procedure for the Move Request. We mentioned before in KMS, the Move request is just an assignment statement.

6. The Store Request

If the request is the Store Request, the store_requests_handler procedure is called by the Kernel_Controller. The following examples illustrate the logic used in this procedure to control the processing of this type of request. Suppose the following CODASYL-DML requests are issued by the user:

```
MOVE SS4 TO PNO IN PA
MOVE PP2 TO SNO IN SA
MOVE PP1 TO SNO IN SP
MOVE SS3 TO PNO IN SP
MOVE 100 TO QTY IN SP
STORE SP
```

KMS translates these CODASYL-DML requests into the following three ABDL RETRIEVE requests and one ABDL INSERT request:

```
[ RETRIEVE ((( TEMPLATE = SP ) and ( SNO = PP1 )) or
        (( TEMPLATE = SP ) and ( PNO = SS3 )))
        ( DBKEY ) BY DBKEY ]

[ RETRIEVE ((TEMPLATE = SA) and (SNO = PP2))(DBKEY) ]

[ RETRIEVE ((TEMPLATE = PA) and (PNO = SS4))(DBKEY) ]

[ INSERT (<TEMPLATE,SP>,<DBKEY,***>,
      <SNO,PP1>,<PNO,SS3>,<QTY,100>,
      <MEMSSP,***>,<MEMPSP,***>) ]
```

The first task performed by the store_requests_handler procedure is to execute the first RETRIEVE request attached to the store_abdl data structure. This request determines if there are records in the database which have attribute values that are not to be duplicate. The execution of this RETRIEVE is accomplished by calling dml_execute. If the request buffer created for this RETRIEVE is non_empty at the end of execution, there is an error. If the request buffer is empty, then we continue the execution in the following manner.

To insert the new record into the correct set occurrences, we need to know the database keys of the owners of the set occurrences. For this reason, we issue the next two RETRIEVE request(s) above. These requests are created by KMS depending on the set selection criteria and the set insertion mode of the new record. These RETRIEVE requests are executed by the dml_execute procedure, and the results are placed in a unique file buffer associated with each request data structure. Then, the build_request is called to complete the INSERT

request. The database keys of the owners of the set occurrences are pulled from the appropriate result buffer(s) and substituted for the place_holding asteriks to complete the INSERT request. In our implementation, the order of the RETRIEVE requests, and the order of the attributes, MEMset_type are the same. Thus, we can easily complete the INSERT request. After this operation, the INSERT request is issued by calling dml_execute. Now, if no currency suppression list is attached to the store_abdl data structure, the CIT table is updated to reflect a change in the SSP and PSP currency, as well as, the current of run_unit. Finally, control returns to KMS via the Kernel_Controller procedure.

7. The Erase Request

The ERASE request is handled by the erase_requests_handler. This procedure first checks the type of the erase request (i.e., ERASE ALL record_type or ERASE record_type). If the type of the erase request is ERASE record_type, then this procedure proceeds in the following manner. The RETRIEVE request attached to the erase_abdl data structure is executed by calling dml_execute. This request determines, whether or not the record being deleted is an owner of a non_empty set. If the request buffer is empty, then all that remains is to issue the DELETE request attached to the erase_abdl data structure. This request deletes the current record of the run_unit. After the deletion, erase_requests_handler update the CIT table by setting the current of run_unit indicator to NULL. If the result buffer of the first RETRIEVE request is not empty, the system gives an error message and the erase request fails.

If the type of the erase request is ERASE ALL record_type. We have a different sequence of events. ERASE ALL deletes the current of run_unit whether or not it is the owner in a non_empty set. Indeed, if it is the owner in a non_empty set, this option really comes into its own. All the members connected to the set are also erased. If any of these members happens to be connnected to some other set of another type, this does not matter. Furthermore, if any of these members happens to be themselves an owner in a non_empty set, then their members in turn are erased. To deal with this problem, the erase_requests_handler procedure calls the erase member procedure recursively.

We first issue the RETRIEVE request to get the database key values of the members of the set. Then, we complete the DELETE request attached to the ret2_node data structure, and issue each DELETE request to KDS via the dml_execute procedure. After this process, the erase_requests_handler executes the DELETE request attached to the erase_abdl data structure to delete the current of the run_unit. Once again, the CIT table is updated to reflect any changes in currency i.e., current of set types become NULL as appropriate, as well as, the current of run unit.

8. The Get Request

The GET request is handled by the dml_kfs procedure. This procedure uses the result buffer of the last FIND request issued. It simply looks at the operation field of the get_node data structure, and retrieves either the entire record or specific fields of the record from the result buffer, and displays these results to the user. We will examine this process more closely in the next chapter (The Kernel Formatting System).

VI. THE KERNEL FORMATTING SYSTEM (KFS)

KFS is the fourth module in the CODASYL-DML language interface, and is called by the Kernel Controller (KC) when it is necessary to display results to the user. The transformation of data into the appropriate format is a very simple task for the CODASYL-DML language interface. Unlike most other language interfaces, no change in format is required. The form that the data is in when it is retrieved from MBDS is the same form in which it is to be displayed to the user. The task of KFS is reduced to simply printing out the results obtained from the ABDL equivalents of the CODASYL-DML requests. But, there is one exception, we do not display the database key values to the user in the result buffer. In this chapter, we discuss how KC stores the data that the KFS eventually displays, and how the KFS outputs this data.

A. THE KFS DATA STRUCTURE

KFS utilizes just one of the data structures defined in the language interface. The kfs_net_info record, shown in Figure 31, contains information needed by KFS to process the results. The first field in this record, response, contains the result from MBDS which is loaded by KC just prior to calling the KFS. The second field, curr_pos, lets KFS know where it is in the response buffer. This assists KFS in maintaining the correct orientation in the response buffer. The last field, res_len, indicates the length of the response buffer. This value is used as a halting condition.

```
struct  kfs_net_info
  {
    char  *response;
    int    curr_pos;
    int    res_len;
  }
```

Figure 31. The kfs_net_info Data Structure.

B. THE FILING OF CODASYL-DML RESULTS

KC stores the results obtained from a CODASYL-DML request by calling the file_results procedure. This procedure first opens the result file for writing in the response. The procedure reads in the name of the first attribute and stores it in a variable, in addition to storing it in the results file. The attribute value is then stored in the results file. A while loop then handles the storing of the remaining attribute-value pairs into the results file. Before an attribute name is stored into the results file, a check is made to determine if this attribute matches the attribute name of the first attribute in the result. If the attribute names match, we have completed storage of one result and are ready to store the next result. An end-of-line marker is inserted in the results file at this point before the next attribute-value pair is stored. Otherwise, the attribute-value pair is stored without the end-of-line marker. This check is one of the reasons that the KFS task of formatting output is so easy for the CODASYL-DML language interface.

C. THE KFS PROCESS

The KFS module is contained in the small procedure, dml_kfs. KFS is only called by KC when the results of a request are to be displayed to the user. The get request causes this action to be taken. The only task that the KFS performs is to display to the screen the attribute-value pair found on the current line in the result buffer of the last FIND request. A loop prints out this line, a character at a time, depending of the type of the get request (i.e., GET or GET item_list IN record_type). This means KFS retrieves either the entire record or specific fields of the record from the result buffer.

VII. CONCLUSION

In this thesis, we have presented the implementation of a CODASYL-DML language interface. This is one of four language interfaces that the multi-lingual database system supports. In other words, the multi-lingual database system is able to execute transactions written in four well-known and important data languages, namely, DL/I, SQL, CODASYL-DML, and Daplex. In our case, CODASYL-DML transactions are executed by way of LIL, KMS, KC and KFS. The work accomplished in this thesis is part of an ongoing research effort being conducted at the Laboratory for Database Systems Research, Naval Postgraduate School, Monterey, California.

The need to provide an alternative to the development of separate stand-alone database systems for specific data models and languages has been the motivation for this research. In this regard, we have shown how a software CODASYL-DML language interface may be constructed without the need of a stand-alone CODASYL database management system. We have extended the work of Wortherly [Ref. 3] by implementing the algorithms he presents for the CODASYL-DML language interface. Additionally, we have provided a general organizational description of the MLDS.

A major design goal has been to design a CODASYL-DML language interface to MBDS without requiring changes to be made to MBDS or ABDL. We have achieved this goal. All CODASYL-DML transactions are performed in the CODASYL-DML interface. MBDS continues to receive and process transactions written in the unaltered syntax of ABDL. Additionally, our implementation does not require any changes to the syntax of CODASYL-DML.

Two facilities suggested by Wortherly [Ref. 3] that are not included in our implementation are the looping facility, PERFORM-LOOP, and the IF-THEN statement. These are not included, due to the lack of time to implement them. Therefore, we chose to concentrate our implementation on the native CODASYL-DML statements first. If more time will become available, we can

implement the these facilities, since there is not logical difficulty in implementing them. Our level-by-level top_down approach to designing the interface has been a fine choice as well. This approach permits follow-on programmers to easily maintain and modify the code.

Once all four interfaces have been completely implemented, MLDS should be tested as a complete system for the projected efficiency, effectiveness, and responsiveness to user needs. It is anticipated that this research and development effort will ultimately result in a new era for database management that will allow for increased productivity in database management.

```
module CODASYL/DML-INTERFACE

    db-list : list;          /* list of existing relational schemas      */
    head-db-list-ptr: ptr;  /* ptr to head of the relational schema list */
    current-ptr: ptr;        /* ptr to the current db schema in the list  */
    follow-ptr: ptr;         /* ptr to the previous db schema in the list */
    db-id : string;          /* string that identifies current db in use  */


proc  LANGUAGE-INTERFACE-LAYER();
    /*  This proc allows the user to interface with the system.  */
    /*  Input and output: user CODASYL-DML requests              */

      stop : int; /* boolean flag */
      answer: char;   /* user answers to terminal prompts */

  perform DML-INIT();
  stop = 'false';
  while (not stop) do
    /* allow user choice of several processing operations */
    print ("Enter type of operation desired");
    print ("    (l) - load new database");
    print ("    (p) - process existing database");
    print ("    (x) - return to the to operating system");
    read (answer);
    case (answer) of
      'l':  /* user desires to load a new database */
            perform LOAD-NEW();
      'p':  /* user desires to process an existing database */
            perform INITIALIZE_CUR_TABLE();
            perform PROCESS-OLD();
      'x':  /* user desires to exit to the operating system */
            /* database list must be saved back to a file   */
            store-free-db-list(head-db-list, db-list);
            stop = 'true';
            exit();
      default: /* user did not select a valid choice from the menu */
               print ("Error - invalid operation selected");
               print ("Please pick again")'
    end-case;
    /* return to main menu */
  end-while;
end-proc;
```

```
proc  DML-INIT();

end-proc;


proc  LOAD-NEW();
    /*  This proc accomplishes the following:                    */
    /*  (1) determines if the new database name already exists,   */
    /*  (2) adds a new header node to the list of schemas,        */
    /*  (3) determines the user input mode (file/terminal),       */
    /*  (4) reads the user input and forwards it to the parser, and */
    /*  (5) calls the routine that builds the template/descriptor files */

    answer: int;        /* user answer to terminal prompts */
    more-input: int;    /* boolean flag */
    proceed: int;       /* boolean flag */
    stop : int;         /* boolean flag */
    db-list-ptr: ptr;   /* pointer to the current database */
    req-str: str;       /* single create in DML form */
    ptr-abdl-list: ptr; /* ptr to a list of ABDL queries (nil for this proc)*/
    tfid, dfid: ptr;    /* pointers to the template and descriptor files */


 /* prompt user for name of new database */
 print ("Enter name of database");
 readstr (db-id);
 db-list-ptr = head-db-list-ptr;
 stop = 'false';
 while (not stop) do
   /* determine if new database name already exists */
   /* by traversing list of network db schemas   */
   if (db-list-ptr.db-id = existing db) then
     print ("Error - db name already exists");
     print ("Please reenter db name");
     readstr (db-id);
     db-list-ptr = head-db-list-ptr;
   end-if;
   else
     if (db-list-ptr + 1 = 'nil') then
       stop = 'true';
     else
       /* increment to next database */
       db-list-ptr = db-list-ptr + 1;
   end-else;
 end-while;
```

```
  /* continue - user input a valid 'new' database name */
  /* add new header node to the list of schemas and fill-in db name */
  /* append new header node to db-list */
  create-new-db(db-id);

  /* the KMS takes the DML defines and builds a new list of relations   */
  /* for the new database. After all of the defines have been processed */
  /* the template and descriptor files are constructed by traversing    */
  /* the new database definition (schema).                              */

  more-input = 'true';
  while (more-input) do
    /* determine user's mode of input */
    print ("Enter mode of input desired");
    print ("    (f) - read in a group of defines from a file");
    print ("    (x) - return to the main menu");
    read (answer);

    case (answer) of
      'f': /* user input is from a file */
           perform READ-TRANSACTION-FILE();
           perform DBD-TO-KMS();
           perform FREE-REQUESTS();
           perform BUILD-DDL-FILES();
           perform KERNEL-CONTROLLER();

      'x': /* exit back to LIL */
           more-input = 'false';

      default: /* user did not select a valid choice from the menu */
           print ("Error - invalid input mode selected");
           print ("Please pick again");
    end-case;
  end-while;

end proc;
```

```
proc  PROCESS-OLD();
    /*  This proc accomplishes the following:                  */
    /*  (1) determines if the database name already exists,    */
    /*  (2) determines the user input mode (file/terminal),    */
    /*  (3) reads the user input and forwards it to the parser */

    answer: int;        /* user answer to terminal prompts */
    found: int;         /* boolean flag to determine if db name is found */
    more-input: int;    /* boolean flag to return user to LIL */
    proceed: int;       /* boolean flag to return user to mode menu */
    db-list-ptr: ptr;   /* pointer to the current database */
    req-str: str;       /* single query in DML form */
    ptr-abdl-list: ptr; /* pointer to a list of queries in ABDL form */
    tfid, dfid: ptr;    /* pointers to the template and descriptor files */


/* prompt user for name of existing database */
print ("Enter name of database");
readstr (db-id);
db-list-ptr = head-db-list-ptr;

found = 'false';
while (not found) do
  /* determine if database name does exist    */
  /* by traversing list of network schemas */
  if (db-id = existing db) then
    found = 'true';
  end-if;
  else
    db-list-ptr = db-list-ptr + 1;
    /* error condition causes end of list('nil') to be reached */
    if (db-list-ptr = 'nil') then
      print ("Error - db name does not exist");
      print ("Please reenter valid db name");
      readstr (db-id);
      db-list-ptr = head-db-list-ptr;
    end-if;

  end-else;

end-while;
```

```
  /* continue - user input a valid existing database name */
  /* determine user's mode of input */

  more-input = 'true';
  while (more-input) do
    print ("Enter mode of input desired");
    print ("    (f) - read in a group of DML requests from a file");
    print ("    (t) - read in a single DML request from the terminal");
    print ("    (x) - return to the previous menu");
    read (answer);

    case (answer) of
      'f': /* user input is from a file */
           perform READ-TRANSACTION-FILE();
           perform DMLREQS-TO-KMS();
           perform FREE-REQUESTS();

      't': /* user input is from the terminal */
           perform READ-TERMINAL();
           perform DMLREQS-TO-KMS();
           perform FREE-REQUESTS();

      'x': /* user wishes to return to LIL menu */
           more-input = 'false';

      default: /* user did not select a valid choice from the menu */
           print ("Error - invalid input mode selected");
           print ("Please pick again");
    end-case;

  end-while;

end-proc;


proc  READ-TRANSACTION-FILE();
  /* This routine opens a dbd/request file and reads the transactions */
  /* into the transaction list.  If open file fails, loop until valid */
  /* file entered                                                     */

  while (not open file) do
    print ("Filename does not exist");
    print ("Please reenter a valid filename");
    readstr ( file);
  end-while;
```

```
proc READ-FILE();
    /* This routine reads transactions from either a file or the */
    /* terminal into the user's request list structure so that   */
    /* each request may be sent to the KERNEL-MAPPING-SYSTEM.    */

end-proc;


proc  READ-TERMINAL();
    /* This routine substitutes the STDIN filename for the read  */
    /* command so that input may be intercepted from the terminal */

end-proc;


proc INITIALIZE-CUR-TABLE();
    /* This proc initialize the CIT table before starting    */
    /* to execute CODASYL-DML request(s).                    */

end-proc;


proc DBD-TO-KMS();
    /* This routine sends the request list of database descriptions */
    /* one by one to the KERNAL-MAPPING-SYSTEM                      */

  while (more-dbds) do
    KERNAL-MAPPING-SYSTEM();
  end-while;

end-proc;
```

```
proc DMLREQS-TO-KMS();
    /* This routine causes the DML requests to be listed to the screen. */
    /* The selection menu is then displayed allowing any of the        */
    /* DML requests to be executed.                                    */

  perform LIST-DMLREQS();
  proceed = 'true';
  while (proceed) do
    print ("Pick the number or letter of the action desired");
    print ("    (num) - execute one of the preceding DML requests");
    print ("    (d)   - redisplay the file of DML requests");
    print ("    (x)   - return to the previous menu");
    read (answer);

    case (answer) of
      'num' : /* execute one of the requests */
              traverse query list to correct query;
              perform KERNAL-MAPPING-SYSTEM();

      'd'   : /* redisplay requests */
              perform LIST-DMLREQS();

      'x'   : /* exit to mode menu */
              proceed = 'false';

      default : /* user did not select a valid choice from the menu */
              print (" Error - invalid option selected");
              print (" Please pick again");
    end-case;

  end-while;

end-proc;
```

```
/*
/*                  Currency Indicator Table
/*   References made in the following specification to CIT refer
/* to the Currency Indicator Table. This table consists of struc-
/* tures that hold information identifying the current record of
/* record_type, set_type, and run_unit (run_unit is the applica-
/* tion program being run). The following is the proposed struc-
/* ture for this table [Ref. 13].
/*
/*       struct  CIT
/*         {
/*          struct  RUN_UNIT       *run;
/*          struct  rec_type_node   *next_rec_type;
/*          struct  set_type_node   *next_set_type;
/*         }
/*
/*       struct  RUN_UNIT
/*         {
/*          char  rec_type[   ];
/*          int   dbkey;
/*         }
/*
/*  For each record type in schema:
/*       struct  rec_type_node
/*         {
/*          char                type[   ];
/*          int                 dbkey;
/*          struct rec_type_node   *next_rec_type;
/*         }
/*
/*
/*  For each set type in schema:
/*       struct  set_type_node
/*         {
/*          boolean              OWNER;
/*          char                 TYPE[   ];
/*          int                  dbkey;
/*          char                 member[   ];
/*          char                 owner[   ];
/*          int                  owner_dbkey;
/*          struct set_type_node   *next_set_type;
%{
   boolean:  first_move = TRUE /* flag for MOVE operation */
   boolean:  first_time     /* general purpose flag */
   boolean:  sys_flag_value /* boolean value of system flags */
```

```
   ptr:  curr_temp_rec      /* ptr to last record added to move_list */
   ptr:  curr_temp_item     /* ptr to next item node to be added to
                            record_template node of movelist */
   list: suppression_list   /* list of record types and/or set types */
                            for which currency updates are suppressed */
   list: select_list        /* list of data items used for record section */
   list: connect_list       /* list of sets to which current of run
                            unit is to be connected or disconnected */
   list: tgt_list           /* list of attribute names to be accessed */
   list: move_list          /* list of record templates used with
                            MOVE statement */
   list:  curr_non_dup_list /* list of data items for which duplicates
                            are not allowed in current record_type */
   int:  level_number       /* level of data item in record types */
   char:  member_type       /* string variable to hold a name */
%}
start  statement

statement: ddl_statement
       | dml
         ;

dml: dml_statement
  | dml dml_statement
    ;

ddl_statement: schema_defn  record_list  set_list
             ;

schema_defn: SCHEMA NAME IS schema_name SEMI_COLON
             {
             locate db_id schema header node
             if (db names do not match)
               print ("Error-given db_name doesn't
                      match name in file")
               perform yyerror()
               return
             end_if
             initialize db_key /* starting value is 1 */
             }
           ;

record_list: record_desc
             {
             set db_id node ndn_first_rec ptr
             }
```

```
        | record_list  record_desc
            {
            connect successive record nodes
            }
          ;

record_desc: record  data_item_list
            {
            curr_non_dup_list = NULL
            }
          ;

record: RECORD  NAME  IS
        {
        allocate and init a new
          record node (NREC_NODE)
        allocate curr_non_dup_list
        db_id_node ndn_num_rec++
        }
      record_spec
      ;

record_spec: record_type
            {
            if (record_type not defined yet)
              copy record_type to current
                record node (NREC_NODE)
              make this the current record node
            end_if
            else
              print ("Error-'record_type' record
                    doubly defined")
              perform yyerror()
              return
            end_else
            }
          SEMI_COLON  duplicates_list
          ;


set_list: empty
      | set_desc
          {
          set db_id node  ndn_first_set ptr
          }
      | set_list set_desc
```

```
        {
        connect successive set node(s)
        }
      ;


set_desc: set_desig  owner_spec  member_spec
      ;

set_desig: SET NAME IS
        {
        allocate and init a new set node (NSET_NODE)
        db_id_node ndn_num_set++
        }
      set_type
        {
        if (set_type not yet defined)
          copy set_type to current set node (nsn_name)
          establish curr_set_ptr
        end_if
        else
          print ("Error-'set_type' set doubly defined in db")
          perform yyerror()
        end_else
        }
      SEMI_COLON
      ;

owner_spec: OWNER IS  aa  SEMI_COLON
        ;

aa: record_type
    {
    if (record_type not defined)
      print ("Error-'record_type' record does not exist")
      perform yyerror()
      return
    end_if
    else
      copy record_type to current set node (nsn_owner_name)
      locate record_type node
      nsn_owner(ptr) = record_type node
    end_else
    }
  | SYSTEM
  ;
```

```
member_spec: MEMBER IS  record_type
             {
             if (record_type not defined)
               print ("Error-'record_type' record does not exist")
               perform yyerror()
               return
             end_if
             else
               copy record_type to current set node (nsn_member_name)
               locate record_type node
               nsn_member(ptr) = record_type node
             end_else
             }
           SEMI_COLON insert_clause retention_clause
             {
             alloc set_select node
             }
           set_select_clause SEMI_COLON
           ;

duplicates_list: empty
             | dupl SEMI_COLON
               ;

dupl: duplicate_spec
    | dupl  duplicate_spec
      ;


duplicate_spec: DUPLICATES ARE NOT ALLOWED FOR item_spec
              ;

item_spec: item_name
           {
           alloc new non_dup node
           copy item_name to non_dup node
           add non_dup node to curr_non_dup_list
           }
         | item_spec COMMA item_name
           {
           alloc successive non_dup nodes
           copy successive item_names to non_dup nodes
           add successive non_dup nodes to curr_non_dup_list
           }
          ;
```

```
data_item_list: item_desc
                {
                connect new attr-node to record_node
                }
              | data_item_list item_desc
                {
                connect successive attr_node(s) to record_node
                }
               ;


item_desc: level_num
           {
           allocate and init a new attr_node (NATTR_NODE)
           NATTR_NODE nan_level_num = level_number
           record_node nrn_num_attr++
           }
         data_item_desc
           {
           if (nan_level_num = level number of current attribute node)
             connect new attr node to current attr node
             if (nan_level_number > 1)
               connect nan_parent ptr of new node
             end_if
           end_if
           else if (nan_level_number > level number of current attr node)
             connect nan_child ptr of current attr node to new attr node
             connect nan_parent ptr of new attr node to current attr node
           end_else_if
           else
             locate last attr node with same level number
             set that node's nan_next_attr ptr to the new attr node
             update current attr pointer
           end_else
           }
         ;
data_item_desc: item_name
                {
                copy item_name to attr_node (NATTR_NODE)
                if (item_name not on curr_non_dup_list)
                  attr_node nan_dup_flag = 1
                end_if
                }
               SEMI_COLON  data_type  PERIOD
               ;
```

```
level_num: empty
          {
          level_number = 1   /* default value */
          }
        | INTEGER
          {
          level_number = INTEGER
          }
        ;


data_type: CHARACTER INTEGER
          {
          attr_node nan_length1 = INTEGER
          attr_node nan_length2 = 0
          attr_node nan_type = 'c'
          }
        | FIXED INTEGER
          {
          attr_node nan_length1 = INTEGER
          attr_node nan_length2 = 0
          attr_node nan_type = 'i'
          }
        | FIXED INTEGER
          {
          attr_node nan_length1 = INTEGER
          }
          INTEGER
          {
          attr_node nan_length2 = INTEGER
          attr_node nan_type = 'f'
          }
        ;


insert_clause: INSERTION IS AUTOMATIC
              {
              set_node nsn_insert = 'a'
              }
```

```
          | INSERTION IS MANUAL
            {
            set_node nsn_insert = 'm'
            }
          ;


retention_clause: RETENTION IS FIXED
                  {
                  set_node nsn_retent = 'f'
                  }
                | RETENTION IS MANDATORY
                  {
                  set_node nsn_retent = 'm'
                  }
                | RETENTION IS OPTIONAL
                  {
                  set_node nsn_retent = 'o'
                  }
                ;


set_select_clause: empty
                   {
                   set_node nsn_select = 'o'
                   }
                 | SEMI_COLON SET SELECTION IS BY  set_select_spec
                   ;

set_select_spec: VALUE OF item_name IN record_type
                 {
                 if(valid_attr(item_name,record_type))
                   copy 'v' to set_select node select_mode
                   copy item_name to set_select node item_name
                   copy record_type to set_select node record1
                   copy BLANK to set_select node record2
                 end_if
                 else
                   print("Error-'item_name' not valid for 'record_type'")
                   perform yyerror()
                   return
                 end_else
                 }
               | STRUCTURAL item_name IN record_type
                 {
                 if(valid_attr(item_name,record_type))
```

```
        copy 's' to set_select node select_mode
        copy item_name to set_select node item_name
        copy record_type to set_select node record1
      end_if
      else
        print("Error-'item_name' not valid for 'record_type'")
        perform yyerror()
        return
      }
      EQ item_name IN record_type
      {
      if(previous item_name equals this item_name)
        if(valid_attr(item_name,record_type))
          copy record_type to set_select node record2
        end_if
        else
          print("Error-'item_name' is not valid for 'record_type'")
          perform yyerror()
          return
      end_if
      else
        print("Error-'item_name' items do not match")
        perform yyerror()
        return
      end_else
      }
    | APPLICATION
      {
      copy 'a' to set_select node select_mode
      copy BLANK to record1, record2, item_name
      }
      ;


dml_statement: set_flag
    | move
    | get
    | find
    | store
    | connect
    | disconnect
    | erase
    | modify
    | perform_loop
    | if_then
      ;
```

```
set_flag: MOVE f_value TO f_name
        ;


f_value: YES
         {
         sys_flag_value = TRUE
         }
       | NO
         {
         sys_flag_value = FALSE
         }
        ;


f_name: EOF
         {
         eof = sys_flag_value
         }
       | NOTFOUND
         {
         notfound = sys_flag_value
         }
        ;

/* The MOVE statement is a COBOL assignment statement that assigns a */
/* value to a particular data field in a record template. We use a   */
/* list structure for this purpose.                                  */

move: MOVE item_value
        {
        if (first_move = TRUE)
          alloc and init move_list
          first_move = FALSE
        end_if
        create new data_item_node
        copy 'item_value' to value field in data_item_node
        establish curr_temp_item pointer
        }
      TO item_name
        {
        copy 'item_name' to name field in data_item_node
        }
```

```
    IN record_type
     {
     if (item_name not in record_type for current schema)
       perform error(2)
       return
     end_if
     else if ('record_type' node on move_list)
           connect curr_temp_item to record_template node
         end_else_if
         else
           create new record_template node
           copy 'record_type' to name field of record_template node
           connect curr_temp_item to record_template node
           add record_template node to move_list
           update curr_temp_rec pointer
         end_else
    ;

/* The GET statement takes the entire current record of the run unit */
/* or specified data fields of the current record of the run unit    */
/* and returns the values to the user.                               */

get: GET
     {
     alloc and init new 'get' node
     select_list = NULL  /* reset select_list */
     }
    mm
     ;


mm: item_list IN record_type
    {
    if ('record_type' is not equal to CIT.RUN_UNIT.type)
      perform error(3)
      return
    end_if
    else
      get_type = ITEMS in get node
      copy record_type to get node
      for (each data_item on item_list)
        if ('data_item' is not defined for record_type)
          perform error(2)
          return
        end_if
        else /* create pseudo tgt_list */
```

```
          copy data_item to get node
        end_else
      end_for
    end_else
    }
| record_type
    {
    if ('record_type' is not equal to CIT.RUN_UNIT.type)
      perform error(3)
      return
    end_if
    else
      get_type = RETURN_ALL in get node
      copy 'record_type' to get node
    end_else
    }
| empty
    {
    get_type = RETURN_ALL in get node
    copy CIT.RUN_UNIT.type to get node
    }
    ;

/* The FIND statements establish the current of run unit, record type, */
/* and set type.                                                       */

find: FIND  record_selection_expr  curr_suppression
    ;
/* The FIND ANY means: find any record of type record_type whose */
/* values for item1 through itemn match those in that record's  */
/* template in the user work area.                              */

record_selection_expr: ANY record_type
                    {
                    if ('record_type' record_template node is not
                        on move_list)
                      perform error(1)
                      return
                    else
                      alloc and init new 'find' node
                      find_type = ANY in find node
                      copy record_type to find node
                      alloc and init new abdl_str
                      alloc and init new tgt_list
                      /* begin forming a RETRIEVE request */
                      copy "[ RETRIEVE ((TEMPLATE = 'record_type')"
```

```
                    to abdl_str
                  end_if
                  select_list = NULL
                  }
                 USING  item_list  IN  record_type
                  {
                  if ('record_type' is same as previous 'record_type')
                    if (any data item on select_list is not
                     defined for record_type)
                     perform error(2)
                     return
                    end_if
                    else
                     create tgt_list item for all attributes
                      of 'record_type' record
                     for (each data item on select_list)
                      if ('data_item' not on move_list)
                       perform error(1)
                       return
                      end_if
                       else
                        get 'item_value' from move_list
                        concat "and ('data_item' = 'item_value')"
                          to abdl_str
                       end_else
                      end_for
                      concat ")('tgt_list') by DBKEY]" to abdl_str
                      connect abdl_str to find node
                    end_else
                  end_if
                  else
                    perform error(6)
                    return
                  end_else
                  }
/* The FIND CURRENT means: Make the current of set_type the current */
/* record of the run unit.                                          */

                | CURRENT record_type WITHIN set_type
                  {
                  if (CIT.set_type.TYPE is not equal to 'record_type')
                    perform error(7)
                    return
                  end_if
                  else
                 /* current of run_unit becomes current of 'set_type' */
```

```
                alloc and init new 'find' node
                find_type = CURRENT in find node
                copy record_type to find node
                copy set_type to find node
                copy CIT.set_type.dbkey to find node
              end_else
              }

/* The FIND DUPLICATE means: Find the first record in the current set_*/
/* type occurrence whose value for item1 through itemn matches those  */
/* for the same items in the current set_type occurrence, not the UWA */
/* record template. This implementation assumes the records being re- */
/* quested are already in a buffer.                                   */

            | DUPLICATE WITHIN set_type
              {
              alloc and init new 'find' node
              find_type = DUPLICATE in find node
              copy set_type to find node
              select_list = NULL  /* reset select_list */
              }
             USING item_list IN record_type
              {
              if ((record_type is not CIT.set_type.TYPE) or
                  (record_type is not CIT.set_type.member))
                perform error(8)
                return
              end_if
              else
                copy record_type to find node
                for (each data_item on select_list)
                  if (any data_item on select_list is not
                    defined for record_type)
                    perform error(2)
                    return
                  end_if
                  else /* create a pseudo tgt_list */
                    copy data_item to find node
                  end_else
                end_for
              end_else
              }
/* This statement means: Find the FIRST, LAST, NEXT, or PRIOR record_*/
/* type record within the current set_type occurrence. The ll token  */
/* takes the value FIRST, LAST, NEXT, or PRIOR.                       */
            | ll  record_type WITHIN set_type
```

```
                {
                if ('record_type' is not a valid member type
                    for 'set_type')
                  perform error(5)
                  return
                end_if
                else
                  copy record_type to find node
                  copy set_type to find node

              /* RETRIEVE all member records of set occurrence */

                  alloc and init new abdl_str
                  alloc and init new tgt_list
                  copy "[RETRIEVE (
                       (TEMPLATE = CIT.set_type.member) and
                       (MEMBER.set_type = CIT.set_type.owner_dbkey))"
                       to abdl_str
                  create tgt_list for all attributes of member record
                  concat "('tgt_list') by DBKEY]" to abdl_str
                  connect abdl_str to find node
                end_else
                }

/* The FIND OWNER means: Find the owner of the current set_type occurrence */

              | OWNER WITHIN set_type
                {
                alloc and init 'find' node
                find_type = OWNER in find node
                copy set_type to find node
                alloc and init new abdl_str
                alloc and init new tgt_list

               /* form RETRIEVE request */

                copy "[RETRIEVE ((TEMPLATE = CIT.set_type.owner)
                     and (DBKEY = CIT.set_type.owner_dbkey))"
                     to abdl_str
                create tgt_list for all attributes of owner record
                concat "('tgt_list')]" to abdl_str
                connect abdl_str to find node
                }
/* This statement means: Find the first record_type record within the */
/* current set_type occurrence whose values for item1 through itemn   */
/* match the values found in the record_type template in the UWA, not */
```

```
/* the values in the current of set_type as in the FIND DUPLICATE.   */

        | record_type WITHIN set_type CURRENT
          {
          if ('record_type' not a member type of 'set_type')
            perform error(5)
            return
          end_if
          else
            alloc and init new 'find' node
            find_type = WITHIN in find_node
            copy record_type to find node
            copy set_type to find node
            alloc and init new abdl_str
            alloc and init new tgt_list

          /* begin forming RETRIEVE request */

            copy "[RETRIEVE ((TEMPLATE = 'record_type') and
                 (MEMBER.set_type = CIT.set_type.owner_dbkey)"
                 to abdl_str
            create tgt_list for all attributes of 'record_type'
              record
            select_list = NULL   /* reset select_list */
          end_else
          }
        USING item_list IN record_type
          {
          if (any data_item on select_list is not defined
              for 'record_type')
            perform error(2)
            return
          end_if
          else if (any data_item on select_list
                   not on move_list)
                 perform error(1)
                 return
          end_else_if
          else
            for (each data_item on select_list)
              get 'item_value' from move_list
              concat "and ('data_item' = 'item_value')
                to abdl_str
            end_for
            concat ")('tgt_list') by DBKEY]" to abdl_str
            connect abdl_str to find node
```

```
                end_else
                }
              ;


ll: FIRST
    {
    alloc and init new 'find' node
    find_type = FIRST in find node
    }
  LAST
    {
    alloc and init new 'find' node
    find_type = LAST in find node
    }
  NEXT
    {
    alloc and init new 'find' node
    find_type = NEXT in find node
    }
  PRIOR
    {
    alloc and init new 'find' node
    find_type = PRIOR in find node
    }
  ;


curr_suppression: LSQUARE supp_expr RSQUARE
          | empty
            ;


supp_expr: SUPPRESS UPDATE
      | UPDATE type_spec
        ;

type_spec: set_type
          {
          add set_type to suppression_list
          }
      | type_spec COMMA set_type
          {
          add successive set_types to suppression_list
          }
        ;
```

```
/* This statement means: Delete the current record of the run unit,    */
/* and all of its descendents regardless of whether they are owners of */
/* other sets.                                                         */

erase: ERASE ALL record_type
       {
       if ('record_type' is not CIT.RUN_UNIT.type)
         perform error(3)
         return
       end_if
       else
         alloc and init new 'erase' node
         erase_type = ALL in erase node
         for (each set_type in schema)
           if (CIT.set_type.owner_dbkey = CIT.RUN_UNIT.dbkey)
             member_type = CIT.set_type.member

             /* form RETRIEVE to get member records */
             alloc and init new abdl_str
             copy"[RETRIEVE(MEMBER.set_type = CIT.RUN_UNIT.dbkey)
                 (DBKEY) by DBKEY]" to abdl_str
             connect abdl_str to erase node

             /* erase member records */
             alloc and init new abdl_str
             copy"[DELETE((TEMPLATE = 'member_type') and
                 (DBKEY = ***))]" to abdl_str
             connect abdl_str to erase node

             /* delete all descendants of member records */
             perform erase_all(member_type,erase node)
           end_if
         end_for

         /* delete current of RUN_UNIT */
         alloc and init new abdl_str
         copy "[DELETE((TEMPLATE = 'record_type') and
              (DBKEY = CIT.RUN_UNIT.dbkey))]" to abdl_str
         connect abdl_str to erase node
       end_else
       }
```

```
/* This statement means: Delete the current record of the run unit if */
/* and only if, it is not the owner of a non-empty set.              */

  | ERASE record_type
    {
    if ('record_type' is not CIT.RUN_UNIT.type)
      perform error(3)
      return
    end_if
    else

      /* erase one record - current of RUN_UNIT */
      alloc and init new 'erase' node
      erase_type = NULL in erase node

      /* form RETRIEVE to see if 'record_type' is */
      /* owner of non_empty set                   */
      alloc and init new abdl_str
      copy "[RETREIVE(" to abdl_str
      first_time = TRUE
      for (each set_type in schema)
        if ('record_type' is owner type of set_type)
          if (first_time)
            concat "(MEMBER.set_type = CIT.RUN_UNIT.dbkey)"
              to abdl_str
            first_time = FALSE
          end_if
          else
            concat "or (MEMBER.set_type = CIT.RUN_UNIT.dbkey)"
              to abdl_str
          end_else
        end_if
      end_for
      concat ")(DBKEY) by DBKEY]" to abdl_str
      connect abdl_str to erase node

      /* for DELETE request */
      alloc and init new abdl_str
      copy "[DELETE ((TEMPLATE = CIT.RUN_UNIT.type) and
           (DBKEY = CIT.RUN_UNIT.dbkey))]" to abdl_str
      connect abdl_str to erase node
    end_else
    }
  ;
```

```
/* The STORE means: Create a new record in the database using values */
/* supplied by the user via MOVE statements, for the data items of   */
/* the specified record_type. The is connected to all sets in which */
/* INSERTION IS AUTOMATIC. The appropriate occurrence of the sets   */
/* must be selected before the new record can be connected. This is */
/* done based on the SET SELECTION clause specified in the database  */
/* schema definition for the sets in question.                       */

store: STORE record_type
        {
        if ('record_ype' record template node is not on move_list)
          perform error(1)
          return
        end_if
        alloc and init new 'store' node
        alloc and init new abdl_str
        copy "[RETRIEVE (" to abdl_str
        first_time = TRUE
        for (each data_item in schema for 'record_type')
          if (nan_dup_flag is set)
            if (data_item in move_list 'record_type' record template)
              get data_item value from move_list
              if (first_time = TRUE)
                concat"((TEMPLATE = 'record_type') and
                        ('data_item' = 'item_value'))" to abdl_str
                first_time = FALSE
              end_if
              else
                concat "or ((TEMPLATE = 'record_type') and
                          ('data_item' = 'item_value'))" to abdl_str
              end_else
            end_if
          end_if
        end_for
        concat")(DBKEY) by DBKEY]" to abdl_str
        connect retrieve request to store node
        alloc and init new abdl_str

                /* Form an INSERT request */
        copy"[INSERT (<TEMPLATE,'record_type'>,<DBKEY,***>" to abdl_str
        for (each 'data_item' in schema for 'record_type')
          if ('data_item' not on move_list for 'record_type')
            perform error(4)
            return
          end_if
          else
```

```
        get data_item value from move_list
        concat",<'item_name','item_value'>" to abdl_str
      end_else
    end_for


  /* Now determine which set occurrences the new record belongs to  */
  /* and add proper attribute-value pairs to the INSERT abdl_str to */
  /* indicate set membership. The following FOR loop and CASE state-*/
  /* ment fill the INSERT abdl_str with the proper pairs.           */
    for (each set_type in schema in which 'record_type' is a member)
      case (set selection mode) of

        /* set selection is by applicaton */
      a: perform by_application(INSERT abdl_str)

        /* set selection is by value */
      v: perform by_value(INSERT abdl_str)

        /* set selection is by structural */
      s: perform by_structural(INSERT abdl_str)

        /* no set selection criteria was given */
      o: perform by_default(INSERT abdl_str)

      end_case
    end_for
    concat "]" to INSERT abdl_str
    connect INSERT abdl_str to store node
    alloc and init suppression_list
    }
    curr_suppression
    {
    connect suppression_list to store node
    }
    ;

/* The MODIFY means: Modify the entire current record of the run unit */
/* or the specified data items in that record.  The new values are    */
/* supplied by the user via the UWA.                                  */

modify: MODIFY
        {
        select_list = NULL   /* reset select_list */
        }
```

```
  item_list IN record_type
    {
    if ('record_type' is not current of run unit)
      perform error(3)
      return
    end_if
    if ('record_type' record_template node is not on move_list)
      perform error(1)
      return
    end_if
    if (any data item on select_list not defined for 'record_type')
      perform error(2)
      return
    end_if
    else
      alloc and init new 'modify' node
      locate record_template node on move_list for 'record_type'
      for (each data_item on select_list)
        alloc and init new abdl_str
        /* form UPDATE request */
        copy "[ UPDATE ((TEMPLATE = 'record_type') and
             (DBKEY = CIT.RUN_UNIT.dbkey))" to abdl_str
        get 'item_value' from move_list
        concat "('data_item' = 'item_value')]" to abdl_str
        connect abdl_str to 'modify' node
      end_for
    end_else
    }
| MODIFY record_type
    {
    select_list = NULL  /* reset select_list */
    if ('record_type' not current of run unit)
      perform error(3)
      return
    end_if
    if ('record_type' record_template node is not on move_list)
      perform error(1)
      return
    end_if
    else
      alloc and init new 'modify' node
      for (each data_item in record_type)
        if (data_item not on move_list for 'record_type')
          perform yyerror(4)
          return
        end_if
```

```
        else
          alloc and init new abdl_str

          /* form an UPDATE request */
          copy "[ UPDATE ((TEMPLATE = 'record_type') and
            (DBKEY = CIT.RUN_UNIT.dbkey)) to abdl_str
          get new 'item_value' from move_list
          concat "('data_item' = 'item_value')]" to abdl_str
          connect abdl_str to 'modify' node
        end_else
      end_for
    end_else
    }
  ;

/* The CONNECT means: Connect the current record of the run unit to the */
/* current occurrence of the specified set type.  There may be several  */
/* sets listed in the statement.                                        */

connect: CONNECT record_type TO
        {
        if ('record_type' is not current of run unit)
          perform error(3)
          return
        end_if
        else
          alloc and init connect_list
        end_else
        }
        set_type_list
        {
        alloc and init 'connect' node
        for (each 'set_type' on connect_list)
          if ('record_type' is not a member type record for 'set_type')
            or (INSERTION is not manual)
            perform error(5)
            return
          end_if
          else
            alloc and init new abdl_str
            copy "[UPDATE ((TEMPLATE = 'record_type') and
                 (DBKEY = CIT.RUN_UNIT.dbkey))
                 (MEMBER.set_type = CIT.set_type.owner_dbkey)]"
                 to abdl_str
            connect new abdl_str to 'connect' node
          end_else
```

```
        end_for
        connect_list = NULL    /* reset connect_list */
        }
      ;  .


set_type_list: set_type
             {
             add 'set_type' to connect_list
             }
           | set_type_list  COMMA  set_type
             {
             add successive 'set_type'(s) to connect_list
             }
            ;

/* The DISCONNECT means: Disconnect the current record of the run unit */
/* from the set type occurrence that contains the record.              */

disconnect: DISCONNECT record_type FROM
           {
           if ('record_type' record is not current record of run unit)
             perform error(3)
             return
           end_if
           else
             alloc and init new connect_list
           end_else
           }
           set_type_list
           {
           alloc and init 'disconnect' node
           for (each set_type on connect_list)
             if ('record_type' is not a member type record for 'set_type')
               perform error(5)
               return
             end_if
             else
               alloc and init new abdl_str
               copy "[UPDATE ((TEMPLATE = 'record_type') and
                    (DBKEY = CIT.RUN_UNIT.dbkey))
                    (MEMBER.set_type = NULL)]"
                    to abdl_str
               connect abdl_str to 'disconnect' node
             end_else
           end_for
```

```
            connect_list = NULL  /* reset connect_list */
            }
          ;


perform_loop: PERFORM UNTIL bb EQ cc
        | END_PERFORM
          ;

bb: EOF
 | NOTFOUND
   ;

cc: YES
 | NO
   ;


item_list: item_name
           {
           put item_name on select_list
           }
       | item_list  COMMA  item_name
           {
           put successive item names on select_list
           }
         ;


schema_name: IDENTIFIER
           ;


record_type: IDENTIFIER
           ;


set_type: IDENTIFIER
        ;


item_name: IDENTIFIER
         ;
```

```
item_value: IDENTIFIER
        | INTEGER
          ;

proc error(err_code)
 /* This procedure prints error messages, causes data structures to */
 /* be de-allocated, and causes proc yyerror to be executed.       */

 case err_code of
  1: print("Error - must initialize 'record_type' record_template")

  2: print("Error - 'data_item' not defined in 'record_type'")

  3: print("Error - 'record_type' is not current record of run unit")

  4: print("Error - attempting to modify or store record without
           giving value of 'data-item'")


  5: print("Error - 'record_type' record does not belong to 'set_type'")

  6: print("Error - record_types specified are not the same")

  7: print("Error - 'record_type' is not current of 'set_type'")

  8: print("Error - 'record_type' must be a member record of 'set_type'")

 end_case
 perform cleanup()   /* free data structures */
 perform yyerror()
 return
end_error;

proc by_application(abdl_str)

  if (set_node nsn_insert ='a' )  /* insertion mode is automatic */
    concat",<MEMBER.set_type,CIT.set_type.owner_dbkey>" to abdl_str
  end_if
  else /* insertion mode is manual */
    concat",<MEMBER.set_type,NULL>" to INSERT abdl_str
  end_else

end_by_application;
```

```
proc by_value(abdl_str)

  locate data_item node in schema for set_select node item_name
    in set_select node record1
  if (nan_dup_flag set)
    get owner record type of set_type from schema
    if (owner record type node not on move_list) or
       (data_item not on move_list)
      perform error(4)
      return
    end_if
    else
      if (set node nsn_insert = 'a')  /* automatic insertion */
        get data_item value from move_list
        copy"[RETRIEVE((TEMPLATE = owner_record_type) and
            (item_name = 'item_value')) (DBKEY)]" to abdl_str
        connect new retrieve request to store node
        concat",<MEMBER.set_type,***>" to INSERT abdl_str
      end_if
      else  /* manual insertion */
        concat",<MEMBER.set_type,NULL>" to INSERT abdl_str
      end_else
    end_else
  end_if /* nan_dup_flag */

end_by_value;



proc by_structural(abdl_str)

  locate data_item nose in schema for set_select node item_name
    in set_select node record1 record_type
  if (nan_dup_flag set)
    get record1 name from set_select node for set_type
    if ('record1' record template node not on move_list) or
       (data_item not on move_list)
      perform error(4)
      return
    end_if
    else
      if (set_node nsn_insert = 'a')  /* automatic insertion */
        get data_item value from move_list
        get record2 name from set_select node for set_type
        copy"[RETRIEVE ((TEMPLATE = record2 name) and
            (item_name = item_value)) (DBKEY)" to abdl_str
```

```
      connect new retrieve request to store node
      concat",<MEMBER.set_type,***>" to INSERT abdl_str
    end_if
    else  /* manual insertion */
      concat",<MEMBER.set_type,NULL>" to INSERT abdl_str
    end_else
  end_else
 end_if  /* nan_dup_flag */

end_by_structural;



proc by_default(abdl_str)

 print("Warning - Attempting to store a record with no
       particular set selection given.  Will assume 'BY
       APPLICATION'")
 if (set_node nsn_insert = 'a')  /* automatic insertion */
  concat",<MEMBER.set_type,CIT.set_type.owner_dbkey>"
    to INSERT abdl_str
 end_if
 else  /* manual insertion */
  concat",<MEMBER.set_type,NULL>" to INSERT abdl_str
 end_else

end_by_default;



proc erase_all(record_type,erase node)

string   member_type;

 for (each set_type in schema)
  if ('record_type' is owner type of set_type)
    member_type = member type of set_type
    /* for RETRIEVE to get members of 'set_type' */
    alloc and init new abdl_str
    copy "[RETRIEVE(MEMBER.set_type = ***)(DBKEY) by DBKEY]"
      to abdl_str
    connect abdl_str to erase node
    /* delete member records */
    alloc and init new abdl_str
    copy "[DELETE((TEMPLATE = 'member_type') and (DBKEY = ***))]"
      to abdl_str
```

```
      connect abdl_str to erase node
      /* erase descendants of member records */
      erase_all(member_type,erase node)
    end_if
  end_for
  return(erase node)

end_erase_all
```

```
module KERNEL_CONTROLLER()

/*  This procedure accomplishes the following:        */
/*  Depending on the dmi_operation the corresponding  */
/*  procedure is called.                              */

{
 case (dmi_operation)
  CreateDB:
      perform LOAD_TABLES();
  Disconnect:
  Connect   :
  ModfItem  :
  ModfRec   :
      perform REQUEST_HANDLER();
  FindReq:
      perform FIND_REQUESTS_HANDLER();
  StorReq:
      perform STORE_REQUESTS_HANDLER();
  EraReq:
      perform ERASE_REQUESTS_HANDLER();
  GetReq:
      perform DML_KFS();
  MoveReq:
      ;
  default:
      /* This handles any errors */
 end_case
}
end_module

LOAD_TABLES()

/* This procedure accomplishes the following:
/* (1) Calls dbl_template which is already
/* defined in the Test Interface. It loads the
/* template file.
/*
/* (2) Calls dbl_dir_tbls() also defined in
/* the Test Interface. It loads the descriptor
/* files.
{
  perform DBL_TEMPLATE();
```

```
  perform DBL_DIR_TBLS();
}
end_proc

REQUEST_HANDLER()

/* This procedure accomplishes the following:
/* (1) Calls dml_execute untill all CODASYL-DML
/* queries associated with abdl_req data
/* structures are processed.
/*
/* (2) If query is Connect or Disconnect, then
/* appropriate action is taken to update the CIT
/* table.
{
 while (! end of request )
   perform DML_EXECUTE(abdl_str,file_ptr);
 if (operation == Connect)
  {
   while (connect_list != NULL)
   {
    current_set_type = run_unit_record_name
    current_set_dbkey = run_unit_dbkey
   }
  }
 if (operation == Disconnect)
  {
   while (connect_list != NULL)
   {
    current_set_type = ' '
    current_set_dbkey = 0
   }
  }
}
end_proc

FIND_REQUESTS_HANDLER()

/* This procedure accomplishes the following:
/* Depending on the find_operation takes
/* appropriate action, then set correct
/* location in the result buffer.
{
 case (find_operation)
     AnyFin:
       perform DML_EXECUTE(abdl_str,file_ptr)
```

```
  if (dont update != TRUE)
   perform Find_update()
FirstFin:
  perform DML_EXECUTE(abdl_str,file_ptr)
  locate first record in the result buffer
  create temporary string to hold this record
  current_buff_val = dbkey value of this record
  increment buffer location
  if (dont update != TRUE)
   perform Find_update()
LastFin:
  perform DML_EXECUTE(abdl_str,file_ptr)
  locate the last record in the result buffer
  create temporary string to hold this record
  current_buff_val = dbkey value of this record
  if (dont update != TRUE)
   perform Find_update()
NextFin:
  first find correct find_node corresponding
  NextFin request
  locate correct result file
  free temporary string
  reload temporary string with the next record value
  current_buff_val = dbkey value of this record
  increment buffer location
  if (dont update != TRUE)
   perform Find_update()
PriorFin:
  first find correct find_node corresponding
  PriorFin request
  locate correct result file
  open this file to read again
  locate prior record of the current record
  free temporary string
  reload temporary string with this record value
  current_buff_val = dbkey value of this record
  decrement buffer location
  if (dont update != TRUE)
   perform Find_update()
OwnerFin:
  perform DML_EXECUTE(abdl_str,file_ptr)
  create temporary string to hold the result
  if (dont update != TRUE)
   perform Find_update()
WithinFin:
  perform DML_EXECUTE(abdl_str,file_ptr)
```

```
        create temporary string to hold the first record
        current_buff_val = dbkey value of this record
        increment buffer location
        if (dont update != TRUE)
         perform Find_update()
      CurFin:
        if (dont update != TRUE)
         perform Find_update()
      DupFin:
        first find correct find_node corresponding
        DupFin request
        locate correct result file
        while (current buffer location <=
            number of result in the result buffer)
         free temporary string
         reload temporary string with this record value
         current_buff_val = dbkey value of this record
         make comparesion to find duplicate record
        end_while
        if (dont_update != TRUE)
         perform Find_update()
  end_case
}
end_proc

FIND_UPDATE()

/* This procedure update the CIT table
/* depending on request
{
 initialize currency table pointer
 case (find_operation)
   AnyFin:
     locate first record in the result buffer
     create temporary string to hold this record
     if (set_list == NULL)
     {
      set set_ptr
     /* determine this record belongs to which set */
     while (set_ptr != NULL)
     {
      if (!on_suppres_list(set_name))
      {
       if (set_owner_name == record_name)
       {
        if (set_name on the CIT table)
```

```
    {
     current_buff_val = dbkey value of this record
     current_set_owner_dbkey = current_buff_val
     current_set_dbkey = current_buff_val
    }
    else
    {
     allocate new cur_set node
     current_buff_val = dbkey value of this record
     current_set_owner_dbkey = current_buff_val
     current_set_dbkey = current_buff_val
    }
   }
  }
 get next set_ptr
 }
}
else
{
 set fset_ptr to set_list of AnyFin node
 while (fset_ptr != NULL)
  store dbkey value(s) of owner set(s) to
  correct field of set_list structure
 end_while
 current_buff_val = dbkey value of this record
 while (set_list != NULL)
 {
  if (!on_suppres_list(set_name))
   {
    if (set_name on the CIT table )
    {
     current_set_owner_dbkey = set_list_dbkey
     current_set_dbkey = current_buff_val
    }
    else
    {
     allocate new cur_set node
     current_set_owner_dbkey = set_list_dbkey
     current_set_dbkey = current_buff_val
    }
   }
  get next set_list
 }
}
/* update current_run_unit */
run_unit_record_name = rec_name of find_ptr
```

```
    run_unit_dbkey = current_buff_val

   PriorFin:
   FirstFin:
   WithinFin:
   LastFin:
   NextFin:
      run_unit_record_name = rec_name of find_ptr
      run_unit_dbkey = current_buff_val
      current_set_type = rec_name of find_ptr
      current_set_dbkey = current_buff_val
   OwnerFin:
      run_unit_record_name =current_set_owner_name
      run_unit_dbkey = current_set_owner_dbkey
      current_set_type = current_set_owner_name
      current_set_dbkey = current_set_owner_dbkey
   CurFin:
      run_unit_record_name = current_set_type
      run_unit_dbkey = current_set_dbkey

  end_case
}
end_proc

STORE_REQUESTS_HANDLER()

/* This procedure executes correct request(s)
/* and update the CIT table
{
 perform(DML_EXECUTE(abdl_str,file_ptr)
 if (!results_are_not_returned)
 {
  print(Error non_duplicate attribute(s) have
        values in the database)
 }
 else
 {
  set member_ptr
  while (member_ptr != NULL)
  {
   if (selection_mode is by VALUE or by STRUCTURAL)
   {
    if (member_ptr->rn_flag == TRUE)
      get next member_ptr
    else if (insertion_mode is AUTOMATIC)
    {
```

```
  perform DML_EXECUTE(abdl_str,file_ptr)
  if (results_are_not_returned)
  {
   print (Error the owner of the set type does
         not have dbkey value)
  }
  else
  {
   perform BUILD_REQUEST(store_ptr,member_ptr)
   get next member_ptr
  }
 }
 else
  ;
 }
 else
  ;
}
if (stored record is a owner record)
 perform BUILD_REQUEST(store_ptr)
perform DML_EXECUTE(store_abdl_str,file_ptr)
} /* end else */
/* update CIT table */
if (dont update != TRUE)
{
run_unit_record_name = rec_name of store_ptr
run_unit_dbkey = last_dbkey -1
if (stored record is not a owner record)
{
 while (member_ptr != NULL)
 {
  if (!on_suppres_list(set_name))
  {
   if (selection_mode is by VALUE or by STRUCTURAL)
   {
    if (insertion_mode is AUTOMATIC)
    {
     if (set_name on the CIT table)
     {
      current_set_type = run_unit_record_name
      current_set_dbkey = run_unit_dbkey
      if (member_ptr->rn_flag == TRUE)
        current_set_owner_dbkey = 0
      else
        current_set_owner_dbkey = current_buff_val
     }
```

```
        else
        {
         allocate new cur_set node
         current_set_dbkey = run_unit_dbkey
         current_set_owner_dbkey = current_buff_val
        }
       }
       else
         ;
      }
      else
      {
       if (insertion_mode is AUTOMATIC)
       {
        current_set_type = run_unit_record_name
        current_set_dbkey = run_unit_dbkey
       }
       else
        ;
       }
      }
      get next member_ptr
      }
    }
  }
}
end_proc

BUILD_REQUEST(store_ptr,member_ptr)

/* This procedure accomplishes the following:
/* Builds an abdl INSERT request in the store
/* template
{
 allocate enough space for store_template
 if (StorFlag != TRUE)
 {
  fill store_template with contents of store_abdl_req
  'til an '*' is hit
  fill store_template with the last dbkey of schema
  increment dbkey value of schema
  skip over the asteriks we just filled with a value
  StorFlag = TRUE
 }
 if ((mem_flag != TRUE) ||
     selection_mode is by APPLICATION ||
```

```
    selection_mode is by OPTIONAL)
 {
  fill store_template with the rest of store_abdl_req
  'til an ' ' is hit
 }
 else
 {
  fill store_template with contents of store_abdl_req
  'til an '*' is hit
  fetch a value from the result file
  put this value into store_template
  fill store_template with the rest of store_abdl_req
  'til an ' ' is hit
 }
 copy store_template to store_abdl_req
}
end_proc

ERASE_REQUESTS_HANDLER()

/* This procedure accomplishes the following:
/* Depending on the erase_operation takes
/* appropriate action
{
 if (erase_operation == RecEra)
  {
   perform DML_EXECUTE(erase_ret_str,file_ptr)
   if (results_are_not_returned)
   {
    perform DML_EXECUTE(erase_abdl_str,file_ptr)
   /* update current run unit */
    run_unit_record_name = ' '
    run_unit_dbkey = 0
   }
   else
   {
    print (Error the record being deleted is an
          owner of a non_empty set)
   }
  }
 else
 {
 /* Erase ALL operation */
  while (member_ptr != NULL)
  {
   perform ERASE_MEMBER(member_ptr)
```

```
    get next member_ptr
   }
   perform DML_EXECUTE(erase_abdl_str,file_ptr)
   /* update current run unit */
   run_unit_record_name = ' '
   run_unit_dbkey = 0
  }
}
end_proc

ERASE_MEMBER(member_ptr)

/* This procedure accomplishes the following:
/* If the record being deleted is an owner of
/* other set, we must delete its member record(s)
{
 perform DML_EXECUTE(member_abdl,file_ptr)
 open correct buffer file to get a value
 /* This is our stopping condition */
 while (all elements in the result buffer have been used)
 {
  pass over attribute name (DBKEY)
  current_buff_val = value of this attribute
  fill member_template with contents of member_delete_req
  'til an '*' is hit
  fetch a value from the result file
  put this value into member_template
  fill member_template with the rest of member_delete_req
  'til an ' ' is hit
  If (this member record being deleted is an
        owner of other set)
  {
   complete retrieve request using current_buff_val
   perform ERASE_MEMBER(member_ptr)
  }
  perform DML_EXECUTE(member_template,file_ptr)
  increment buffer location
 }
 perform free_cur_set(set_name)
}
end_proc

FREE_CUR_SET(set_name)

/* This procedure accomplishes the following:
/* Frees given set type on the CIT table
```

```
{
 initialize currency set_ptr
 current_flag = FALSE
 while ((set_ptr != NULL) && (current_flag == FALSE))
 {
  if (set_ptr->name == set_name)
  {
   free(set_ptr)
   current_flag = TRUE
  }
  if (current_flag != TRUE)
   get next set_ptr
 }
}
end_proc

DML_EXECUTE(string,file_ptr)

/* This procedure accomplishes the following:
/* (1) Sends the request to MBDS using
/* TI_S$TrafUnit() which is defined in the Test
/* Interface
/*
/* (2) Calls dml_check_requests_left() to ensure
/* that all requests have been received
{
 perform TI_S$TrafUnit(string)
 perform dml_check_requests_left(file_ptr)
}
end_proc

DML_CHECK_REQUESTS_LEFT(file_ptr)

/* This procedure accomplishes the following:
/* (1) Receives the message from MBDS by calling
/* TI_R$Message() which is defined in the Test
/* Interface
/*
/* (2) Gets the message type by calling
/* TI_R$Type
/*
/* (3) If not all the responses to the request
/* have been returned, a loop is entered. Within
/* this loop a case statement separates the
/* responses received by message type
/*
```

```
/* (4) If the response contained no errors,
/* then procedure TI_R$Req_res() is called to
/* receive the response from MBDS
/*
/* (5) If no results are returned, then
/* the boolean results_are_not_returned is set
/* to TRUE
/*
/* (6) If the message contained an error,
/* then procedure TI_R$ErrorMessage is called
/* to get the error message and then procedure
/* TI_ErrRes_output is called to output the
/* error message
{
 results_are_not_returned = FALSE
 done = FALSE
 while ( !done)
 {
  TI_R$Message()
  msg_type = TI_R$Type()
  case (msg_type)
    CH_ReqRes:
      done = TI_R$Req_res(&rid,response)
      if (string length of (response) == 0)
        results_are_not_returned = TRUE
      else
      {
       t = file_results(file_ptr)
       if ( t > max length of previous results ).
        max_length = t
      }
    ReqsWithErr:
     /* handle error conditions */
  }
}
end_proc

FILE_RESULTS(file_ptr)

/* This procedure accomplishes the following:
/* (1) Opens a file to place the results in
/*
/* (2) Keep track of how many results have
/* been received
/*
/* (3) Puts the results in their own line
```

```
/*
/* (4) Returns the length of the response
{
 /* Next two statements are initialization */
 initialize buff_loc
 initialize num_values
 /* If this is the first time then we open
    file for write status   */
 if (file_ptr->nfi_status == FIRSTTIME)
 {
  perform init_buffer()
  open file for write mode
  set nfi_status to RESTTIME
  buff_loc = buff_loc + 1
 }
 else
  open file for append status
 res_len = string length of (response)
 curr_pos = 1
 /* Read first attribute from response */
  read_dml_response(first_attr,curr_pos)

 /* Put this attribute in buffer */
  put_in_buff(first_attr)

 /* Read the value corresponding to this attribute */
  read_dml_response(temp_str,curr_pos)

 /* Put this value in the buffer */.
  put_in_buff(temp_str)
  save_max = curr_pos

 /* Increment the count of values */
  num_values = num_values + 1

 /* While we are not at the end of the response */
 while (curr_pos < (res_len - 2))
 {
  read_dml_response(temp_str,curr_pos)
 /* If the attribute name just read in is not the same
    as the first attribute name proviously read in,
    then we put it and its value on the same line in
    the buffer as the first attribute */
  if (first_attr != temp_str)
  {
   put_in_buff(temp_str)
```

```
  read_dml_response(temp_str,curr_pos)
  put_in_buff(temp_str)
  max_char = curr_pos
  max_char2 = max_char - cur_length
  if (max_char2 > save_max)
   save_max = max_char2
 }
/* If they are the same, then we need to start a new
   line in the buffer    */
 else
 {
  cur_length = curr_pos - val_len
  put_in_buff("0)
  put_in_buff(temp_str)
  read_dml_response(temp_str,curr_pos)
  put_in_buff(temp_str)
  num_values = num_values + 1
 }
}
close file
open file
return(save_max)
}
end_proc

DML_KFS()

/* This procedure accomplishes the following:
/* Pulls all attributes or some specific
/* attribute(s) of a record and displays
/* it to the user
{
first find correct get node
if (get operation == GetItem)
{
 while ( i <= number of attributes of this record)
 {
  pull specific attributes and its values from
  temp_str of net_file_info structure
  display them to the user
 }
}
else
{
while ( i <= number 0f attributes of this record)
   display temp_str to the user
```

```
 }
}
end_proc

PUT_IN_BUFF(string)

/* This procedure accomplishes the following:
/* Puts the incoming string form file_results
/* into the correct file buffer

end_proc

INIT_BUFFER()

/* This procedure accomplishes the following:
/* (1) Copies the user's ID name into a temp
/* string
/*
/* (2) Converts the current dmi_buff_count to
/* a string
/*
/* (3) Increments the above count to reflect
/* the fact that the next time this procedure
/* is called it initialize a new buffer
/*
/* (4) strcat above count to temp
/*
/* (5) strcat BUFF_FILE_SUFFIX to temp
/*
/* (6) strcpy temp over to nfi_buff.fi_fname

end_proc

READ_DML_RESPONSE(outstr,pos)

/* This procedure accomplishes the following:
/* Reads the next value of the response buffer
{
 load outstr with the contents of response until
 an End Marker is detected
 put a ' ' in outstr
}
end_proc
```

```
ON_SUPPRES_LIST(set_name)

/* This procedure accomplishes the following:
/* Checks the given set on the suppres_list or not
{
 /* set correct supp_ptr */
 if (dmi_operation == FindReq)
   set supp_ptr
 else
   set supp_ptr
 while (supp_ptr != NULL)
 {
  if (supp_ptr->set_name == set_name)
   return(TRUE)
  get next supp_ptr
 }
 return(FALSE)
}
end_proc
```

APPENDIX D - THE CODASYL-DML USERS' MANUAL

A. OVERVIEW

The CODASYL-DML language interface allows the user to input transactions from either a file or the terminal. A transaction may take the form of either database descriptions of a new database, or CODASYL-DML requests against an existing database. Database descriptions may only be input from a file, while CODASYL-DML requests may be input from either a file or the terminal. The CODASYL-DML language interface is menu-driven. When the transactions are read from either a file or the terminal, they are stored in the interface. If the transactions are database descriptions, they are executed automatically by the system. If the transactions are CODASYL-DML requests, the user is prompted by another menu to selectively choose an individual CODASYL-DML request to be processed. The menus provide an easy and efficient way to allow the user to view and select the methods in which to process CODASYL-DML transactions. Each menu is tied to its predecessor, so that by exiting each menu the user is moved up the "menu tree". This allows the user to perform multiple tasks in a single session.

B. USING THE SYSTEM

There are two operations the user may perform. The user may either define a new database or process requests against an existing database. The first menu displayed prompts the user for an operation to perform. This menu, hereafter referred to as MENU1, looks like the following:

```
Enter type of operation desired
    (l) - load a new database
    (p) - process old database
    (x) - return to the operating system

ACTION ----> _
```

Upon selecting the desired operation, the user is prompted to enter the name of the database to be used. When loading a new database, the database name provided may not presently exist in the database schema. Likewise, when processing requests against an existing database, the database name provided has to exist in the present database schema. In either case, if an error occurs, the user is told to rekey a different name. The session continues once a valid name is entered.

If the "p" operation is selected from MENU1, a second menu is displayed that asks for the mode of input. This input may come from a data file or interactively from the terminal. This generic menu, MENU2, looks like the following:

```
Enter mode of input desired
    (f) - read in a group of transactions from a file
    (t) - read in transactions from the terminal
    (x) - return to the previous menu

ACTION ----> _
```

If users wish to read transactions from a file, they are prompted to provide the name of the file that contains those transactions. If users wish to enter transactions directly from the terminal, a message is displayed reminding them of the correct format and special characters that are to be used.

If the "l" operation is selected from MENU1, a second menu is displayed that is identical to MENU2 except that the "t" option is omitted. Since the transaction list stores both database descriptions and CODASYL-DML requests, two different access methods have to be employed to send the two types of transactions to the KMS. Therefore, our discussion branches to handle the two processes the user may encounter.

1. Processing Database Descriptions

When the user has specified the filename of the schema, further user intervention is not required. It does not make sense to process only a single schema definition statement out of a set of schema definition statements that produce a new database, since they all have to be processed at once and in a specific order. Therefore, the mode of input is limited to files, and the

transaction list of schema definition statements is automatically executed by the system. Since all the schema definition statements have to be sent at once to form a new database, control should not return to MENU2 where further transactions may be input. Instead, control returns to MENU1 where the user may select a new operation or a new database to process against.

2. Processing CODASYL-DML Requests

In this case, after users have specified the mode of input, they conduct an interactive session with the system. First, all CODASYL-DML requests are listed to the screen. As the CODASYL-DML requests are listed from the transaction list, a number is assigned to each CODASYL-DML request in ascending order starting with the number one. The number is printed on the screen beside the first line of each CODASYL-DML request. Next, an access menu, called MENU3, is displayed which looks like the following:

```
Pick the number or letter of the action desired
    (num) - execute one of the preceding CODASYL-DML requests
    (d)   - redisplay the list of CODASYL-DML requests
    (x)   - return to the previous menu

ACTION ----> _
```

Since the displayed CODASYL-DML requests may exceed the vertical height of the screen, only a full screen of CODASYL-DML requests are displayed at one time. If the desired CODASYL-DML request is not displayed on the current page, the user may depress the RETURN key to display the next page of CODASYL-DML requests. If the user only desires to display a certain number of lines, after the first page is displayed the user may enter a number, and only that many lines of CODASYL-DML requests are displayed. If users are only looking for certain CODASYL-DML requests, once they have found them, they do not have to page through the entire transaction list. By depressing the "q" key, control is broken from listing CODASYL-DML requests, and MENU3 is displayed. Under normal conditions, when the end of the transaction list has been viewed, MENU3 appears.

Since CODASYL-DML requests are independent items, the order in which they are processed does not matter. The users have the choice of executing however many CODASYL-DML requests they desire. A loop causes the transaction list and MENU3 to be redisplayed after each CODASYL-DML request has been executed so that further choices may be made. Unlike processing the schema definition, control returns to MENU2 since the user may have more than one file of CODASYL-DML requests against a particular database, or the user may wish to input some extra CODASYL-DML requests directly from the terminal. Once the user is finished processing on this particular database, the user may exit back to MENU1 to either change operations or exit to the operating system.

C. DATA FORMAT

When reading transactions from a file or the terminal, there has to be some way of distinguishing when one transaction ends and the next begins. Transactions are allowed to span multiple lines, as evidenced by a typical multi-level MOVE statements followed by a STORE statement. This example also shows that our definition of transaction incorporates one or more requests. This allows a group of logically related requests to be executed as a group. Since the system is reading the input line by line, an end-of-transaction flag is required. In our system this flag is the "@" character. Likewise, the system needs to know when the end of the input stream has been reached. In our system the end-of-file flag is represented by the "$" character. The following is an example of an input stream with the necessary flags that are required when multiple transactions are entered:

```
TRANSACTION #1
@
TRANSACTION #2
REQUEST #1
REQUEST #2
.
:

REQUEST #n
@
TRANSACTION #3
@
.
:

@
TRANSACTION #n
$
```

D. RESULTS

When the results of the executed transactions are sent back to the user's screen, they are displayed exactly the same way individual CODASYL-DML requests are displayed. The following consolidates the user's options:

KEY	FUNCTION
return	Displays next screenful of output
(number)	Displays only (number) lines of output
q	Stops output, MENU1 is then redisplayed

LIST OF REFERENCES

1. Demurjian, S. A. and Hsiao, D. K., "New Directions in Database-Systems Research and Development," in the *Proceedings of the New Directions in Computing Conference,* Trondheim, Norway, August, 1985; also in Technical Report, NPS-52-85-001, Naval Postgraduate School, Monterey, California, February 1985.
2. Banerjee, J. and Hsiao, D.K., "A Methodology for Supporting Existing CODASYL Databases with New Database Machines", *Proceedings of National ACM Conference,* 1978.
3. Wortherly, C. R. *The Design and Analysis of a Network Interface for the Multi-Lingual Database System* , M.S. Thesis, Naval Postgraduate School, Monterey, California, Dec 1985.
4. Hsiao, D. K., and Harary, F., "A Formal System for Information Retrieval from Files," *Communications of the ACM,* Vol. 13, No. 2, February 1970, also in Corrigenda, Vol 13., No. 4, April 1970.
5. Wong, E., and Chiang, T. C., "Canonical Structure in Attribute Based File Organization," *Communications of the ACM,* September 1971.
6. Rothnie, J. B. Jr., "Attribute Based File Organization in a Paged Memory Environment," *Communications of the ACM,* Vol. 17, No. 2, February 1974.
7. The Ohio State University, Columbus, Ohio, Technical Report No. OSU-CISRC-TR-77-4, *DBC Software Requirements for Supporting Network Databases,* by J. Banerjee, D. K. Hsiao,and Douglas S. Kerr November 1977.
8. Naval Postgraduate School, Monterey, California, Technical Report, NPS52-85-002, *A Multi-Backend Database System for Performance Gains, Capacity Growth and Hardware Gains,* by S. A. Demurjian, D. K. Hsiao and J. Menon, February 1985.
9. Date, C. J., *An Introduction to Database Systems* , 3d ed., pp. 389-446, Addison Wesley, 1982.
10. Boehm, B. W., *Software Engineering Economics,* Prentice-Hall, 1981.
11. Naval Postgraduate School, Monterey, California, Technical Report, NPS52-84-012, *Software Engineering Techniques for Large-Scale Database Systems as Applied to the Implementation of a Multi-Backend Database System,* by Ali Orooji, Douglas Kerr and Daivid K. Hsiao, August 1984.
12. The Ohio State University, Columbus, Ohio, Technical Report No. OSU-CISRC-TR-82-1, *The Implementation of a Multi Backend Database System (MDBS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MDBS,* by D. S. Kerr et al, January 1982.
13. Kernighan, B. W., and Ritchie, D. M., *The C Programming Language,* Prentice-Hall, 1978.

14. Howden, W. E., "Reliability of the Path Analysis and Testing Strategy," *IEEE Transactions on Software Engineering,* Vol. SE-2, September 1976.

15. Johnson, S. C., *Yacc: Yet Another Compiler-Compiler,* Bell Laboratories, Murray Hill, New Jersey, July 1978.

16. Lesk, M. E. and Schmidt, E., *Lex - A Lexical Analyzer Generator,* Bell Laboratories, Murray Hill, New Jersey, July 1978.

17. Benson, T. P. and Wentz, G. L., *The Design and Implementation of a Hierarchial Interface for the Multi-Lingual Database System* M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

18. Meyer, G. and MacDougal, P., *An Attribute_value Translation of CODASYL's Data Manipulation Language,* Ohio State University, 1982.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2. Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
4. Curricular Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5100	1
5. Professor David K. Hsiao, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	2
6. Steven A. Demurjian, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	2
7. Turk Deniz Kuvvetleri Egitim Daire Baskanligi Bakanliklar Ankara TURKEY	5
8. Bogazici Universitesi Bilgisayar Muhendisligi Bebek Istanbul TURKEY	1
9. Ahmet Emdi Gazi Kemal Mah. Firin Sok. No. 19 Babaeski/Kirklareli - TURKEY	1

10. Bulent Emdi 2
Gazi Kemal Mah.
Firin Sok. No. 19
Babaeski/Kirklareli - TURKEY

The implementation of a network CODASYL-DML interface for the multi-lingual data-base system.